valFORTH™
SOFTWARE SYSTEM
GENERAL UTILITIES

Strings

UMOVE	(addr1 addr2 n --)	UMOVE is a "universal" memory move. It takes the block of memory n bytes long at addr1 and copies it to memory location addr2. UMOVE correctly uses either CMOVE or <CMOVE.
" ccc"	(--)	(at compile time)
	(-- addr)	(at run time) If compiling, the sequence ccc (delimited by the trailing ") is compiled into the dictionary as a string: | len | c | c | c |...| c |
SCONSTANT	xxx ($ --	(at compile time)
	xxx: (-- $)	(at execution time) Takes the string on top of the stack and compiles it into the dictionary with the name xxx. When xxx is later executed, the address of the string is pushed onto the stack.
SVARIABLE	xxx (n --)	Reserves space for a string of length n.
	xxx: (-- $)	When xxx is later executed, the address of the string is pushed onto the stack.
S.	($ --)	Takes the string on top of the stack and sends it to the current output device.
S!	($ addr --)	Takes the string at second on stack and stores it at the address on top of stack.
$+	($1 $2 -- $3)	Takes $2 and concatenates it with $1, leaving $3 at PAD.
LEFT$	($1 n -- $2)	Returns the leftmost "n" characters of $1 as $2.
RIGHT$	($1 n -- $2)	Returns the rightmost "n" characters of $1 as $2.
MID$	($1 n u -- $2)	Returns $2 of length u starting with the nth character of $1.
LEN	($ -- len)	Returns the length of the specified string.
ASC	($ -- c)	Returns the ASCII value of the first character of the specified string.
$COMPARE	($1 $2 -- flag)	Compares $1 with $2 and returns a status flag.
$=	($1 $2 -- flag)	Compares two strings on top of the stack.
$<	($1 $2 -- flag)	Compares two strings on top of the stack.
$>	($1 $2 -- flag)	Compares two strings on top of the stack.
SAVE$	($1 -- $2)	As most string operations leave resultant strings at PAD, the word SAVE$ is used to temporarily move strings to PAD+512.
INSTR	($1 $2 -- n)	Searches $1 for first occurrence of $2. Returns the character position in $1 if a match is found; otherwise, zero is returned.
CHR$	(c -- $)	Takes the character "c" and makes it into a string of length one and stores it at PAD.
DSTR$	(d -- $)	Takes the double number d and converts it to its ASCII representation as $ at PAD.
STR$	(n -- $)	Takes the single length number n and converts it to its ASCII representation as $ at PAD.
STRING$	(n $1 -- $2)	Creates $2 as n copies of the first character of $1.
#IN$	(n -- $)	#IN$ has three similar but different functions. If n is positive, it accepts a string of n or fewer characters from the terminal. If n is zero, it accepts up to 255 characters from the terminal. If n is negative, it returns only after accepting -n characters from the terminal. The resultant string is stored at PAD.
IN$	(-- $)	Accepts a string of up to 255 characters from the terminal.
$-TB	($1 -- $2)	Removes trailing blanks from $1 leaving new $2.
$XCHG	($1 -- $2)	Exchanges the contents of $1 with $2.

Array Word Glossary

ARRAY	xxx (n --)	(compiling)
	xxx:(m -- addr)	(executing) When compiling, creates an array named xxx with n 16-bit elements numbered 0 thru n-1. Initial values are undefined. When executing, takes an argument, m, off the stack and leaves the address of element m of the array.
CARRAY	xxx (n --)	(compiling)
	xxx:(m -- addr)	(executing) When compiling, creates a c-array named xxx with n 8-bit elements numbered 0 thru n-1. Initial values are undefined. When executing, takes an argument, m, off the stack and leaves the address of element m of the c-array.
TABLE	xxx (--)	(compiling)
	xxx:(m -- addr)	(executing) When compiling, creates a table named xxx but does not allot space. Elements are compiled in directly with , (comma). When executing, takes one argument, m off the stack and, assuming 16-bit elements, leaves the address of element m of the table.
CTABLE	xxx (--)	(compiling)
	xxx:(m -- addr)	(executing) When compiling, creates a c-table named xxx but does not allot space. Elements are compiled in directly with C, (c-comma). When executing, takes one argument, m off the stack and, assuming 8-bit elements, leaves the address of element m of the c-table.
VECTOR	xxx (n0 ... nN count ..)	(compiling)
	xxx:(m -- addr)	(executing) When compiling, creates a vector named xxx with count 16-bit elements numbered 0-N. n0 is the initial value of element 0, nN is the initial value of element N, and so on. When executing, takes one argument, m, off the stack and leaves the address of element m on the stack.
CVECTOR	xxx (b0 ... bN count --)	(compiling)
	xxx:(m -- addr)	(executing) When compiling, creates a c-vector named xxx with count 8-bit elements numbered 0-N. b0 is the initial value of element 0, bN is the initial value of element N, and so on. When executing, takes an argument, m, off the stack and leaves the address of element m on the stack.

Double Number Extensions

DVARIABLE	xxx (d --)	At compile time, creates a double number
	xxx:(-- addr)	variable xxx with the initial value d. At run time, xxx leaves the address of its value on the stack.
DCONSTANT	xxx (d --)	At compile time, creates a double number
	xxx:(-- d)	constant xxx with the initial value d. At run time, xxx leaves the value d on the stack.
D-	(d1 d2 -- d3)	Leaves d1-d2=d3.
D0=	(d -- flag)	If d is equal to 0. leaves true flag; otherwise, leaves false flag.
D=	(d1 d2 -- flag)	If d1 equals d2, leaves true flag; otherwise, leaves false flag.
D0<	(d -- flag)	If d is negative, leaves true flag; otherwise, leaves false flag.
D<	(d1 d2 -- flag)	If d1 is less than d2, leaves true flag; otherwise, leaves false flag.
D>	(d1 d2 -- flag)	If d1 is greater than d2, leaves true flag; otherwise, leaves false flag.
DMIN	(d1 d2 -- d3)	Leaves the minimum of d1 and d2.
DMAX	(d1 d2 -- d3)	Leaves the maximum of d1 and d2.
D>R	(d --)	Sends the double number at top of stack to the return stack.
DR>	(-- d)	Pulls the double number at top of the return stack to the stack.
D,	(d --)	Compiles the double number at top of stack into the dictionary.
DU<	(ud1 ud2 -- flag)	If the unsigned double number ud1 is less than the unsigned double number ud2, leaves a true flag; otherwise, leaves a false flag.
M+	(d1 n -- d2)	Converts n to a double number and then sums with d1.

High Resolution Text Output

GCINIT	(--)	Initializes the graphic character output routines. This must be executed prior to using any other hi-res output words.
GC.	(n --)	Displays the single length number n at the current hi-res cursor location.
GC.R	(n1 n2 --)	Displays the single length number n1 right-justified in a field n2 graphic characters wide. See .R .
GCD.R	(d n --)	Displays the double length number d right-justified in a field n graphic characters wide. See D.R .
GCEMIT	(c --)	Displays the text character c at the current hi-res cursor location. Three special characters are interpreted by GCEMIT.
GCLEN	(addr n -- len)	Scans the first n characters at addr and returns the number of characters that will actually be displayed on screen.
GCR	(--)	Repositions the hi-res cursor to the beginning of the next hi-res text line. See CR .
GCLS	(--)	Clears the hi-res display and repositions the cursor in the upper lefthand corner.
GCSPACE	(--)	Sends a space to the graphic character output routine. See SPACE .
GCSPACES	(n --)	Sends n spaces to the graphic character output routine. See SPACES .
GCTYPE	(addr n --)	Sends the first n characters at addr to the graphic character output routine. See TYPE .
GC" ccc"	(--)	Sends the character string ccc (delimited by ") to the graphic character output routine.
GCBKS	(--)	Moves the hi-res cursor back one character position for overstriking or underlining.
GCPOS	(horz vert --)	Positions the hi-res cursor to the coordinates specified. Note that the upper lefthand corner is 0,0.
GC$.	(addr --)	Sends the string found at addr and preceded by a count byte to the graphic character output routine. See $. .
SUPER	(--)	Forces the graphic character output routine into the superscript mode (or out of the sub-script mode). See VMI below. May be performed within a string by the ^ character.
SUB	(--)	Forces the graphic character output routine into the subscript mode (or out of the super-script mode). See VMI below. May be performed within a string by the v character.
VMI	(n --)	The VMI command sets the number of eighths of characters to scroll up or down when either a SUPER or SUB command is issued.
VMI#	(-- addr)	A variable set by VMI.
OSTRIKE	(ON or OFF --)	If the OSTRIKE option is ON, characters are printed over top of the previous characters giving the impression of overstriking.
GCBAS	(-- addr)	A variable which contains the address of the character set displayed by GCEMIT. To change character sets, simply store the address of your new character set into this variable.
GCLFT	(-- addr)	A variable which holds the column position of the left margin.
GCRGT	(-- addr)	A variable which holds the column position of the right margin.

Case Structures

CASE: structure

Format:

```
CASE:  wordname
       word0
       word1
       ...
       wordN  ;
```

CASE Structure

Format:

```
: wordname
  ...
  CASE
     word0
     word1
     ...
     wordN
( NOCASE  wordnone )
  CASEND
  ... ;
```

SEL Structure

Format:

```
: wordname
  ...
  SEL
     n1  ->  word0
     n2  ->  word1
     ...
     nN   >  wordN
( NOSEL  wordnone  )
  SELEND
  ... ;
```

COND Structure

Format:

```
: wordname
  ...
  COND
     condition0  <<  words0  >>
     condition1  <<  words1  >>
     ...
     conditionN  <<  wordsn  >>
( NOCOND  wordsnone  )
  CONDEND
  ... ;
```

Miscellaneous Utilities

XR/W	(#secs addr blk flag --)	"Extended read-write." The same as R/W except that XR/W accepts a sector count for multiple sector reads and writes. Starting at address addr and block blk, read (flag true) or write (flag false) #secs sectors from or to disk.
LOADS	(start count --)	Loads count screens starting from screen # start.
THRU	(start finish -- start count)	Converts two range numbers to a start-count format.
SEC	(n --)	Provides an n second delay. Uses a tuned do-loop.
MSEC	(n --)	Provides an n millisecond delay. (approx) Uses a tuned do-loop.
H->L	(n1 -- n2)	Moves the high byte of n1 to the low byte and zero's the high byte, creating n2. Machine code.
L->H	(n1 -- n2)	Moves the low byte of n1 to the high byte and zero's the low byte, creating n2. Machine code.
H/L	(n1 -- n1(hi) n1(lo))	Split top of stack into two stack items: New top of stack is low byte of old top of stack. New second on stack is old top of stack with low byte zeroed.
BIT	(b -- n)	Creates a number n that has only its bth bit set. The bits are numbered 0-15.
?BIT	(n b -- f)	Leaves a true flag if the bth bit of n is set. Otherwise leaves a false flag.
TBIT	(n1 b -- n2)	Toggles the bth bit of n1, making n2.
SBIT	(n1 b -- n2)	Sets the bth bit of n1, making n2.
RBIT	(n1 b -- n2)	Resets the bth bit of n1, making n2.
STICK	(n -- horz vert)	Reads the nth stick (0-3) and resolves the setting into horizontal and vertical parts, with values from -1 to +1. -1 -1 means up and to the left.
PADDLE	(n1 -- n2)	Reads the n1th paddle (0-7) and returns its value n2. Machine code.
16TIME	(-- n)	Returns a 16 bit timer reading from the system clock at locations 19 and 20, decimal.
8RND	(-- b)	Leaves one random byte from the internal hardware. Machine code.
16RND	(-- n)	Leaves one random word from the internal hardware. Machine code with 20 cycle extra delay for rerandomization.
CHOOSE	(u1 -- u2)	Randomly choose an unsigned number u2 which is less than u1.
CSHUFL	(addr n --)	Randomly rearrange n bytes in memory, starting at address addr.
SHUFL	(addr n --)	Randomly rearrange n words in memory, starting at address addr.
DUMP	(addr n --)	Starting at addr, dump at least n bytes (even multiple of 8) as ASCII and hex. May be exited early by pressing a CONSOLE button.
BXOR	(addr count b --)	Starting at address addr, for count bytes, perform bit-wise exclusive OR with byte b at each address.
BAND	(addr count b --)	Starting at address addr, for count bytes, perform bit-wise AND with byte b at each address.
BOR	(addr count b -)	Starting at address addr, for count bytes, perform bit-wise OR with byte b at each address.
STRIG	(n -- flag)	Reads the button of joystick n (0-3).
PTRIG	(n -- flag)	Reads the button of paddle n (0-7).

Placement of Tabs for Utilities/Editor Documentation

The tab titles included should be cut apart and inserted into the tabs in the following order, starting at the fifth position:

* valFORTH ED. 1.1 — Locate before section XI
* $-ARY-CASE-DBL — Locate before section XII
* HRT-MSC-TRNS — Locate before section XIII

VALPAR
INTERNATIONAL
3801 E. 34TH STREET
TUCSON, ARIZONA 85713
602-790-7141

valFORTH™
SOFTWARE SYSTEM
for ATARI*

GENERAL UTILITIES AND VIDEO EDITOR

*Atari is a trademark of Atari, Inc., a division of Warner Communications

valFORTH T.M.

Screen Oriented Video Editor

Version 1.1
March 1982

The FORTH language is a very powerful addition to the Atari home computer. Programs which are impossible to write in BASIC (usually because of limitations in speed and flexibility) can almost always be written in FORTH. Even when one has mastered the BASIC language, making corrections or additions to programs can be tedious. The video editor described here removes this problem from the FORTH environment. Similar to the MEMO PAD function in the Atari operating system, this editor makes it possible to insert and delete entire lines of code, insert and delete single characters, toggle between insert and replace modes, move entire blocks of text, and much more.

valFORTH™
SOFTWARE SYSTEM

GENERAL UTILITIES AND VIDEO EDITOR

Stephen Maguire
Evan Rosen

VALPAR INTERNATIONAL

Disclaimer of Warranty
on Computer Programs

All Valpar International computer programs are distributed on an "as is" basis without warranty of any kind. The total risk as to the quality and performance of such programs is with the purchaser. Should the programs prove defective following their purchase, the purchaser and not the manufacturer, distributor, or retailer assumes the entire cost of all necessary servicing or repair.

Valpar International shall have no liability or responsibility to a purchaser, customer, or any other person or entity with respect to any liability, loss, or damage caused directly or indirectly by computer programs sold by Valpar International. This disclaimer includes but is not limited to any interruption of service, loss of business or anticipatory profits or consequential damages resulting from the use or operation of such computer programs.

Defective media (diskettes) will be replaced if diskette(s) is returned to Valpar International within 30 days of date of sale to user.

Defective media (diskettes) which is returned after the 30 day sale date will be replaced upon the receipt by Valpar of a $12.00 Replacement Fee.

valFORTH UTILITIES/EDITOR USER'S MANUAL

Table of Contents

GENERAL UTILITIES and VIDEO EDITOR

Overview

This editor is a powerful extension to the valFORTH system designed specifically for the Atari 400/800 series of microcomputers. The main purpose for this editor is to give the FORTH programmer an easy method of text entry to screens for subsequent compilation. The editor has four basic modes of operation:

1) It allows entering of new text to a FORTH screen as though typing on a regular typewriter.

2) It allows quick, painless modification of any text with a powerful set of single stroke editing commands.

3) It pinpoints exactly where a compilation error has occurred and sets up the editor for immediate correction and recompilation.

4) Given the name of a precompiled word, it locates where the original text definition of the word is on disk, if the "LOCATOR" option had been selected when the word was compiled.

The set of single stroke editing commands is a superset of the functions found in the MEMO PAD function of the standard Atari operating system. In addition to cursor movement, single character insertion/deletion, and line insertion/deletion, the editor supports a clear-to-end-of-line function, a split command which separates a single line into two lines, and a useful insert submode usually found only in higher quality word processors.

In addition, there are provisions for scrolling both forwards and backwards through screens, and to save or "forget" any changes made. This is useful at times when text is mistakenly modified.

Also provided is a visible edit storage buffer which allows the user to move, replace, and insert up to 320 lines of text at a time. This feature alone allows the FORTH programmer to easily reorganize source code with the added benefit of knowing that re-typing mistakes are avoided. Usage has shown that once edit-buffer management is learned, significant typing and programming time can be saved.

For those times when not programming, the editor can double as a simple word processor for writing letters and filling other documentation needs. The best method for learning how to use this powerful editor is to enter the edit mode and try each of the following commands as they are encountered in the reading.

As stated above, there are four ways in which to enter the video editor. The following four commands explain each of the possibilities. Note that the symbol "<ret>" indicates that the "RETURN" key is to be typed.

V view screen (scr# ---)

To edit a screen for the first time, the "View" command is to be used. The video display will enter a 32 character wide mode and will be broken into three distinct sections. For example,

50 V <ret>

should give something like the display shown in fig. 1.

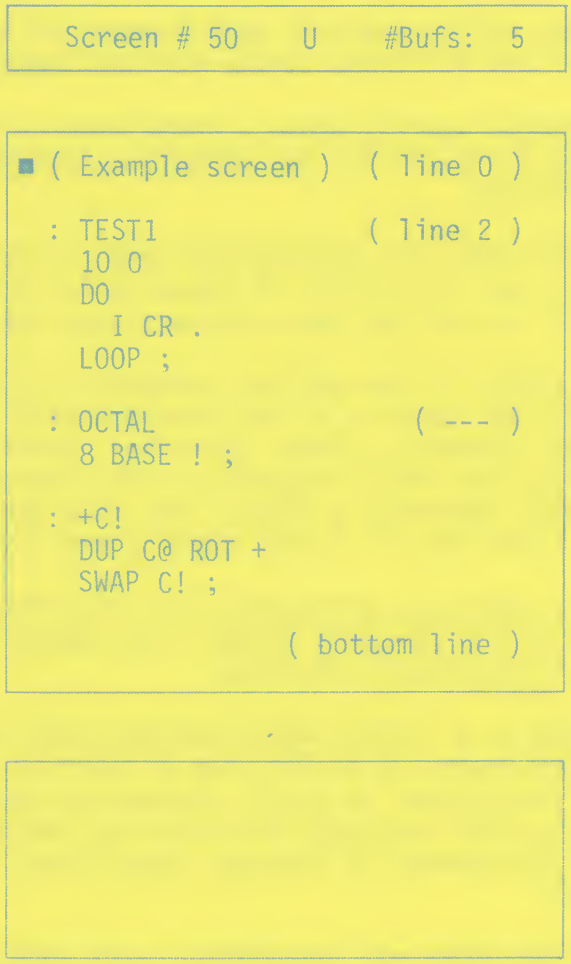


Fig. 1

The top window, composed of a single line, indicates in decimal which screen is currently being edited. One should always make a practice of checking this screen number to insure that editing will be done on the intended screen. Often times, when working with other number bases, the wrong screen is called up accidentally and catching this mistake early can save time. Also shown is the size of the edit buffer (described later). In this example, the buffer is five lines in length. This window is known as the heading window.

FORTH screens typically are 1K (1024 characters) long. Since it is impossible to see an entire screen simultaneously, this editor reveals only half a screen at a time. There is an "upper" half and a "lower" half. In the center of the heading window, either a "U" or an "L" is displayed indicating which half of the current screen is being viewed. If the valFORTH system is in the half-K screen mode, neither "U" nor "L" is displayed since an entire half-K screen can be viewed at one time. In figure 1, the upper half of a full-K screen is being viewed.

The second window (the text window) contains the text found on the specified screen. This window is 32 characters wide and 16 lines high. The white cursor (indicated by the symbol "■") will be in the upper-lefthand corner of the screen awaiting editing commands.

The final five-line window found at the bottom of the screen is known as the buffer window. This is used for advanced editing and is described in greater detail in the section entitled "Buffer Management."

L — re-edit last screen — (---)

This command is used to re-edit the "Last" screen edited. It functions identically to the "V" command described above, except no screen number is specified.

Example: L <ret> (re-edit screen 50)

WHERE — find location of error — (---)

If, when compiling code, a compilation error occurs, the WHERE command will enter the edit mode and position the cursor over the last letter of the offending word. The word can then be fixed and the screen can be re-compiled. Bear in mind that using the WHERE command prior to any occurrence of an error could give strange results.

LOCATE — locate definition — cccc (---)

Once source text has been compiled into the dictionary, it loses easy readability to all but experts of the FORTH language. Often times, though, it is helpful to see what the original source code was. The DECOMP command found in the debugger helps tremendously in this regard, however, some structures such as IF and DO are still difficult to follow. For this reason, the LOCATE command is included with the editor.

This command accepts a word name, and if at all possible it will actually direct the editor to load in the screen where that word was defined. This is very helpful at times when one cannot remember where the original text was. If the screen shown in figure 1 were loaded and the command

LOCATE +C! <ret>

were given, the editor would call up screen 50 and position the cursor over the word ":" which is the beginning of the definition for "+C!". Typically, the LOCATE command will point to ":" , "CODE" , "CONSTANT" , and other defining words.

There is a drawback to this feature, however. In order to call up any word, the LOCATE command must know where the word actually is. Normally, when a word is compiled, there is no way of knowing where it was loaded from. Thus for the LOCATE command to work, each time a word is entered into the dictionary, three extra bytes of memory must be used to store this lookup information. For an application with many words, these extra bytes per word add up quickly, and this is not always desirable. For this reason, the LOCATOR command (described below) allows the user to enable or disable the storage of this lookup information. Only words that were compiled with the LOCATOR option selected can be located. If a word cannot be located, the user is warned, or if the DEBUGGER is loaded, the word is DECOMPed giving pseudo original code.

LOCATOR enable/disable location (ON/OFF ---)

In order for a word to be locatable using LOCATE, the LOCATOR option must have been selected prior to compiling the word. The LOCATOR option is selected by executing "ON LOCATOR" and deselected by executing "OFF LOCATOR". For example:

```
ON LOCATOR
: PLUS   ." = " + . ;              (partial view of a screen)
: STAR   42 EMIT ;
OFF LOCATOR
: NEGATE  MINUS ;
```

Only the words PLUS and STAR can be located. NEGATE cannot be located since the LOCATOR option was disabled. If the DEBUGGER were loaded, NEGATE would be decompiled (see the debugger), otherwise, the user would be given a warning. The default value for LOCATOR is OFF.

#BUFS set buffer length (#lines ---)

The #BUFS command allows the user to specify the length (in terms of number of lines) of the special edit storage buffer. The power of the edit buffer lies in the number of lines that can be stored in it. Although the default value is five, practice shows that at least 16 lines should be set aside for this buffer. The maximum number of lines allowable is 320 which is enough to hold 20 full screens simultaneously.

The following sections give a detailed description of all commands which the video editor recognizes. A quick reference command list can be found following these descriptions.

Cursor Movement

When the edit mode is first entered via the "V" command, a cursor is placed in the upper lefthand corner of the screen. It should appear as a white block and may enclose a black letter. Whenever any key is typed and it is not recognized as an editor command, it is placed in the text window where the cursor appears. Likewise, any line functions (such as delete line) work on the line where the cursor is found.

ctrl∧ , ctrl∨ , ctrl < , ctrl > move-cursor commands

To change the current edit line or character, one of four commands may be given. These are known as cursor commands. They are the four keys with arrows on them. These keys move the cursor in the direction specified by the arrow on the particular key pressed. There are times, however, when this is not the case.

If the current cursor line is the topmost line of the text window, and the "cursor-up" command is issued (by simultaneously typing "ctrl" and "up-arrow"), the cursor will move to the bottom line of the text window. Likewise, a subsequent "cursor-down" command would return the cursor to the topmost line of the window. Similarly, if the cursor is positioned on the leftmost edge and the "cursor-left" command is given, the cursor will "wrap" to the rightmost character ON THE SAME LINE. Issuing "cursor-right" will wrap back to the first character on that line.

RETURN next-line command

Normally, the RETURN key positions the cursor on the first character of the next line. If RETURN is pressed when the cursor is on the last line of the text window (i.e., when the last text line of the screen is current), the cursor is positioned in the upper lefthand corner of the screen.

TAB tabulate command

The TAB key is used to tabulate to the next fixed four column tabular stop to the right of the current cursor character. TABbing off the end of the current line simply places the cursor at the beginning of that same line.

NOTE:

Many commands in the editor will "mark" a current FORTH screen as updated so that any changes made can be preserved on disk. As simple cursor movement does not change the text window in any way, these commands never mark the current FORTH screen. See the section on screen management for more information.

Editing Commands

Editing commands are those commands which modify the text in some predefined manner and mark the current FORTH screen as updated for later saving.

ctrl INS — character insert command

When the "insert-character" command is given, a blank character is inserted at the current cursor location. The current character and all characters to the right are pushed to the right by one character position. The last character of the line "falls off" the end and is lost. The inserted blank then becomes the current cursor character. This is the logical complement to the "delete-character" command described below.

ctrl DEL — delete character command

When the "delete-character" command is issued, the current cursor character is removed, and all characters to the right of the current cursor character are moved left one position, thus giving a "squeeze" effect. This is normally called "closing" a line. The rightmost character on the line (which was vacated) is replaced with a blank. This serves as the logical complement to the "insert-command" described above.

shift INS — line insert command

The "line-insert" command inserts a blank line between the current cursor line and the line immediately above it. The current line and all lines below it are moved down one line to make room for the new line. The last line on the screen falls off the bottom and is lost. If this command is accidentally typed, the "oops" command (ctrl-O) described later can be used to recover from the mistake. Also see the "from buffer" command described in the section on buffer management for a similar command. This command serves as the logical complement to the "line-delete" command described below.

shift DEL — line delete command

The "line-delete" command deletes the current cursor line. All lines below the current line are brought up one line and a blank line fills the vacated bottom line of the text window. The deleted line is lost. If this command is accidentally issued, recovery can be made by issuing the "oops" command (ctrl-O) described later. Also see the "to-buffer" command described in the section on buffer management for a similar command. The "delete-line" command serves as the logical complement to the "line-insert" command.

ctrl H — erase to end of line

The "Hack" command performs a clear-to-end-of-line function. The current cursor character and all characters to the right of it on the current line are blank filled. All characters blanked are lost. The "oops" command described later can be used to recover from an accidentally hacked line.

ctrl I — insert/replace toggle

In normal operation, any key typed which is not recognized by the editor as a control command will replace the current cursor character with itself. This is the standard replace mode. Normally, if one wanted to insert a character at the current cursor location, the insert character command would have to be issued before any text could be entered. If inserting many characters, this is cumbersome.

When active, the insert submode automatically makes room for any new characters or words and frees the user from having to worry about this. When the editor is called up via the "V" command, the insert mode is deactivated. Issuing the insert toggle command will activate it and the cursor will blink, indicating that the insert mode is on. Issuing the command a second time will deactivate the insert mode and restore the editor to the replace mode. Note that while in the insert mode, all edit commands (except BACKS, below) function as before.

BACKS — delete previous character

The BACKS key behaves in two different ways, depending upon whether the editor is in the insert mode or in the replace mode. When issued while in the replace mode, the cursor is backed up one position and the new current character is replaced with a blank. If the cursor is at the beginning of the line, the cursor does not move, but the cursor character is still replaced with a blank.

If the editor is in the insert mode, the cursor backs up one position, then deletes the new current cursor character and then closes the line. If the cursor is at the beginning of the line, the cursor remains in the same position, the cursor character is deleted and the line closed.

NOTE:

As all of the above commands modify the text window in some manner, the screen is marked as having been changed. This is to be sure that all changes made are eventually saved on disk. The "quit" command described in the section on changing screens allows one to unmark a screen so that major mistakes need not be saved.

Buffer Management

Much of the utility of the valFORTH editor lies in its ability to temporarily save text in a visible buffer. To aid the user, it is possible to temporarily send text to the buffer and to later retrieve it. This storage buffer can hold as many as 320 lines of text simultaneously. This buffer is viewed through a 5 line "peephole" visible as the last window on the screen. Using this buffer, it is possible to duplicate, move, and easily reorganize text, in addition to temporarily saving a line that is about to be edited so that the original form can be viewed or restored if necessary. The following section will explain exactly how to accomplish each of these actions.

ctrl T — to buffer command

The "to-buffer" command deletes the current cursor line, but unlike the "delete-line" command where the line is lost, this command moves the "peephole" down and copies the line to the bottom line of the visible buffer window. This line is the current buffer line. The buffer is rolled upon each occurrence of this command so that it may be used repeatedly without the loss of stored text.

For example, if the cursor is positioned on line eight of the display shown in figure 1 and the "to-buffer" command is issued twice, the final result will be as shown in figure 2.

ctrl F — from buffer command

The "from-buffer" command does exactly the opposite of the "to-buffer" command described above. It takes the current buffer line and inserts it between the current cursor line and the line above it. The cursor line and all lines below it are moved down one line with the last line of the text window being lost. If the cursor were placed on line 14 of the above screen display and the "from-buffer" command were issued once, the display in figure 3 would result.

```
 Screen # 50    U    #Bufs: 5
```

```
( Example screen )  ( line 0 )

: TEST1                ( line 2 )
  10 0
  DO
    I CR .
  LOOP ;

■
: +C!
  DUP C@ ROT +
  SWAP C! ;

                  ( bottom line )
```

Current:

```


: OCTAL                  ( --- )
  8 BASE ! ;
```

Current:

fig. 2

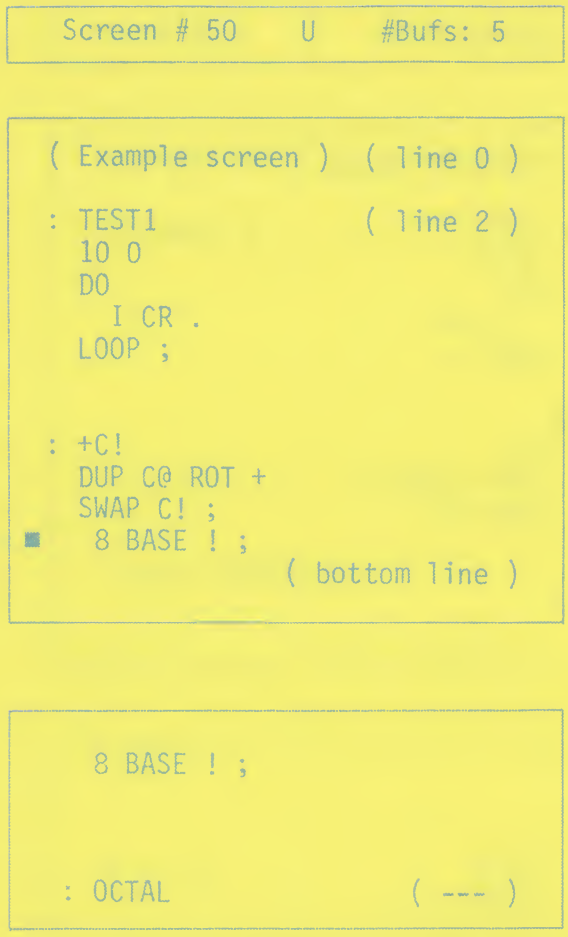


fig. 3

If the "from-buffer" command is issued again, then lines 13 through 15 of the text window would look like:

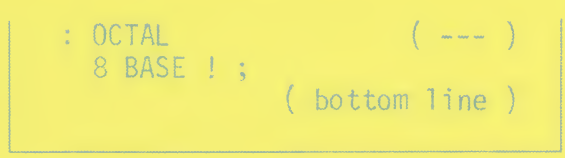


fig. 4

Note that a block of text has been moved on the screen. Larger blocks of text can be moved in the same manner.

ctrl K copy to buffer command

The "copy-to-buffer" command takes the current cursor line and duplicates it, sending the copy to the buffer. This commands functions identically to the "to-buffer" command described above, except that the current cursor line is NOT deleted from the text window.

ctrl U copy from buffer

The "copy-from-buffer" command replaces the current cursor line with the current buffer line. This command functions identically to the "from-buffer" command described above, except that the buffer line is not inserted into the text window, it merely replaces the current cursor line. The "oops" command described below can be used to recover from accidental usage of this command.

ctrl R roll buffer

The "roll-buffer" command moves the buffer "peephole" down one line and redisplays the visible window. If the buffer were the minimum five lines in length, the bottom four lines in the window would move up a line and the top line would "wrap" to the bottom and become the current buffer line. If there were more than five buffer lines, the bottom four lines would move up a line, the topmost line would be pushed up behind the peephole, and a new buffer line coming up from below the peephole would be displayed and made current. For example, if the buffer were five lines long and contained:

```
                ( Who?   )
                ( What?  )
                ( When?  )
                ( Where? )
Current:        ( Why?   )
```

Fig. 5

the "roll-buffer" command gives:

```
                ( What?  )
                ( When?  )
                ( Where? )
                ( Why?   )
Current:        ( Who?   )
```

Fig. 6

ctrl B back-roll-buffer command

The "back-roll-buffer" does exactly the opposite of the "roll-buffer" command described above. For example, if given the buffer in figure 6 above, the "back-roll" command would give the buffer shown in figure 5.

ctrl C clear buffer line command

The "clear-buffer-line" command clears the current buffer line and then "back-rolls" the buffer so that successive clears can be used to erase the entire buffer.

NOTE:

Any of the above commands which change the text window will mark the current screen as updated. Those commands which alter only the buffer window (such as the "roll" command) will not change the status of the current screen.

Changing Screens

There are four ways in which to leave a FORTH screen. These four methods are: moving to a previous screen, moving to a following screen, saving the current screen and exiting, or simply aborting the edit session. The four commands allowing this are now described:

ctrl P — previous screen command

The "previous-screen" command has two basic functions. If the lower part of the current screen is being viewed in the text window, this command simply displays the upper portion of the screen. If the upper portion is already being viewed, then the "previous-screen" command saves any changes made to the current screen and then loads in the screen immediately before the current screen. The lower part of the screen will then be displayed. If in the half-K screen mode, however, this command simply changes screens.

ctrl N — next screen command

Like the "previous-screen" command described above, the "next-screen" command also has two basic functions. If the upper part of a screen is being viewed, this command simply displays the lower portion. If, on the other hand, the lower part of the screen is being edited, any changes made to the current screen are saved and the next screen is loaded.

ctrl S — save command

The "save" command saves any changes made to the current screen and exits the edit mode. The video screen is cleared, and the number of the screen just being edited is displayed for reference. Note that it is usually a good idea to immediately FLUSH (described in the section on screen management below) any unsaved screens.

ctrl Q — quit command

The "quit" command aborts the edit session "forgetting" any changes made to the text visible in the text window. Changes made on previously edited screens will NOT be forgotten. The "quit" command is usually used when either the wrong screen has been called up, or if it becomes desirable to start over and re-edit the screen again.

Special Commands

There are four special commands in this editor which allow greater flexibility in programming on the valFORTH system:

ESCAPE — special key command

The "special-key" command instructs the video editor to ignore any command function of the key typed next and force a character to the screen. For example, normally when "ctrl >" is typed, the cursor is moved right. By typing "ESCAPE ctrl >" the cursor is not moved -- rather, the right-arrow is displayed.

ctrl A — arrow command

When dealing with FORTH screens, it is often necessary to put the FORTH word "-->" (pronounced "next screen") or the valFORTH word "==>" (pronounced "next screen") or the valFORTH word "==>" (pronounced "next half-K screen") at the end of a screen for chaining a long set of words together. This command automatically places, or erases, an arrow in the lower right hand corner of the text window. If "-->" is already there, it is replaced with "==>". If "==>" is found, it is erased. (This command marks the screen as updated.)

ctrl J — split line command

Often times, for formatting reasons, it is necessary to "split" a line into two lines. The split line command takes all characters to the left of the cursor and creates the first line, and with the remaining characters of the original line, a second line is created. Graphically, this looks like:

```
before:   | The quick■brown fox jumped.        |


after:    | The quick■                         |
          |          brown fox jumped.         |
```

Since a line is inserted, the bottom line of the text window is lost. Using the "oops" command below, however, this can be recovered.

ctrl O — oops command

Occasionally, a line is inserted or deleted accidentally, half a line cleared by mistake, or some other major editing blunder is made. As the name implies, the "oops" command corrects most of these major editing errors. The "oops" command can be used to recover from the following commands:

1)	insert line command	(shift INS)
2)	delete line command	(shift DEL)
3)	hack command	(ctrl H)
4)	to buffer command	(ctrl T)
5)	from buffer command	(ctrl F)
6)	copy from buffer command	(ctrl U)
7)	split line command	(ctrl J)

Screen Management

In addition to the commands available while in the edit mode, there are several other commands which are for use outside of the edit mode. Typically, these commands deal with entire screens at a time.

FLUSH (---)

When any changes are made to the current text window, the current screen is marked as having been changed. When leaving the edit mode using the "save" command, the current screen is sent to a set of internal FORTH buffers. These buffers are not written to disk until needed for other data. Thus, if no other screen is ever accessed, the buffers will never be saved to disk. The FLUSH command forces these buffers to be saved if they have been marked as being modified.

Example: FLUSH <ret>

EMPTY-BUFFERS (---)

Occasionally, screens are modified temporarily or by accident, and get marked as being modified. The EMPTY-BUFFERS command unmarks the internal FORTH buffers and fills them with zeroes so that "bad" data are not saved to disk. Zero filling the buffers ensures that the next access to any of the screens that were in the buffers will load the unadulterated copy from disk. The abbreviation MTB is included in the valFORTH system to make the use of this command easier.

Examples: EMPTY-BUFFERS <ret>
MTB <ret>

COPY (from to ---)

To duplicate a screen, the COPY command is used. The screen "from" is copied to the screen "to" but not flushed.

Example: 51 60 COPY <ret>

(Copies screen 51 to screen 60.)

CLEAR (scr# ---)

The CLEAR command fills the specified screen with blanks so that a clean edit can be started. The screen is then made current so that the L command can be used to enter the edit mode.

Example: 50 CLEAR <ret>

(Clears screen 50 and makes it current.)

CLEARS (scr# #screens ---)

The CLEARS command is used to clear blocks of screens at a time. After user verification, it starts with the specified screen and clears the specified number of consecutive screens. The first screen cleared is made current so that the L command can be used to enter the edit mode.

```
Example:     25  3  CLEARS    <ret>
             Clear from SCR 25
                    to SCR 27  <Y/N> Y
```

(Screens 25-27 are cleared. Screen 25 is made current.)

SMOVE (from to #screens ---)

The SMOVE command is a multiple screen copy command used for copying large numbers of consecutive screens at a time. User verification is required by this command to avoid disastrous loss of data. All screens to be copied are read into available memory and the user is prompted to initiate the copy. This allows the swapping of disks between moves to make disk transfers possible. The number of screens the SMOVE command can copy at a time is limited only by available memory.

```
Example:     50  60  5  SMOVE    <ret>
             SMOVE from 50 thru 54
                    to 60 thru 64  <Y/N> Y
             Insert source <RETURN>  <ret>
             Insert dest. <RETURN>  <ret>
```

(Transfers the specified screens.)

Editor Command Summary

Below is a quick reference list of all the commands which the video editor recognizes.

Entering the Edit Mode: (executed outside of the edit mode)

V (scr# ---)
Enter the edit mode and view the specified screen.

L (---)
Re-view the current screen.

WHERE (---)
Enter the edit mode and position the cursor over the word that caused a compilation error.

LOCATE cccc (---)
Enter the edit mode and position the cursor over the word defining "cccc".

LOCATOR (ON/OFF ---)
When ON, allows all words compiled until the next OFF to be locatable using the LOCATE command above.

#BUFS (#lines ---)
Sets the length (in lines) of the storage buffer. The default is five.

Cursor Movement: (issued within the edit mode)

ctrl ^	Move cursor up one line, wrapping to the bottom line if moved off the top.
ctrl v	Move cursor down one line, wrapping to the top line if moved off the bottom.
ctrl <	Move cursor left one character, wrapping to the right edge if moved off the left.
ctrl >	Move cursor right one character, wrapping to the left edge if moved off the right.
RETURN	Position the cursor at the beginning of the next line.
TAB	Advance to next tabular column.

Editing Commands: (issued within the edit mode)

ctrl INS	Insert one blank at cursor location, losing the last character on the line.
ctrl DEL	Delete character under cursor, closing the line.
shift INS	Insert blank line above current line, losing the last line on the screen.
shift DEL	Delete current cursor line, closing the screen.
ctrl I	Toggle insert-mode/replace-mode. (see full description of ctrl-I).
BACKS	Delete last character typed, if on the same line as the cursor.
ctrl H	Erase to end of line (Hack).

Buffer Management: (issued within the edit mode)

ctrl T	Delete current cursor line sending it to the edit buffer for later use.
ctrl F	Take the current buffer line and insert it above the current cursor line.
ctrl K	Copy current cursor line sending it to the edit buffer for later use.
ctrl U	Take the current buffer line and copy it to the current cursor line.
ctrl R	Roll the buffer making the next buffer line current.
ctrl B	Roll the buffer backwards making the previous buffer line on the screen current.
ctrl C	Clear the current buffer line and perform a ctrl-B.

Note: The current buffer line is last line visible on the video display.

Changing Screens: (issued within the edit mode)

ctrl P	Display the previous screen saving all changes made to the current text window.
ctrl N	Display the next screen saving all changes made to the current text window.
ctrl S	Save the changes made to the current text window and end the edit session.
ctrl Q	Quit the edit session forgetting all changes made to current text window.

Special Keys: (issued within the edit mode)

ESC	Do not interpret the next key typed as any of the commands above. Send it directly to the screen instead.
ctrl A	Put "-->", "==>", or erase the lower right-hand corner of the text window.
ctrl J	Split the current line into two lines at the point where the cursor is.
ctrl O	Corrects any major editing blunders.

Screen Management: (executed outside of the edit mode)

FLUSH (---)

Save any updated FORTH screens to disk.

EMPTY-BUFFERS (---)

Forget any changes made to any screens not yet FLUSHed to disk. Used in "losing" major editing mistakes. The abbreviation MTB is more commonly used.

COPY (from to ---)

Copies screen #from to screen #to.

CLEAR (scr# ---)

Blank fills specified screen. This performs the same functions as "WIPE" in Leo Brodie's book.

CLEARS (scr# #screens ---)

Blank fills the specified number of screens starting with screen scr#.

SMOVE (from to #screens ---)

Duplicate the specified number of screens Starting with screen number "from". Allows swapping of disks before saving screens to screen number "to".

STRING UTILITIES

The following collection of words describes the string utilities of the valFORTH Utilities Package. Strings have been implemented in the FORTH language in many different ways. Most implementations set aside space for a third stack -- a string stack. As strings are entered, they are moved (using CMOVE) to this stack. When strings are manipulated on this stack, many long memory moves are usually required. This method is typically much slower than the method implemented in valFORTH.

Rather than waste memory space with a third stack, valFORTH uses the already existing parameter stack. Unlike the implementation described above, valFORTH does not store strings on the stack. Rather, it stores the addresses of where the strings can be found.* Using this method, words such as SWAP , DUP , PICK , and ROLL can be used to manipulate strings. Routines such as string sorts which work on many strings at a time are typically much faster since addresses are manipulated rather than long strings. In practice, we have found few if any problems using this method of string representation.

String Glossary

For the purposes of this section, a string is defined to be a sequence of up to 255 characters preceded by a byte indicating its length. The first character of the string is referenced as character one. If the length of the string is zero, it has no characters and is called the "null" string. In stack notation, strings are represented by the symbol $ and the address of the string is stored on the stack rather than the string itself*.

-TEXT addr1 n addr2 -- flag

The word -TEXT compares n characters at address1 with n characters at address2. Returns a false flag if the sequences match, true if they don't. Flag is positive if the character sequence at address1 is alphabetically greater than the one at address2. Flag is zero if the character sequences match, and is negative if the character sequence at address1 is alphabetically less than the one at address2.

-NUMBER addr -- d

-NUMBER functions identically to the standard FORTH word NUMBER with the only difference being that -NUMBER does not abort program execution upon an illegal conversion. -NUMBER takes the character string at addr and attempts to convert it to a double number. On successful conversion, the value d is returned with the status variable NFLG set to one. On unsuccessful conversion, a double number zero is returned with the variable NFLG set to zero. -NUMBER is pronounced "not number".

*Representing strings on the stack by their addresses is a very useful concept borrowed from MMS Forth (TRS-80), authored by Tom Dowling, and available from Miller Microcomputer Services, 617-653-6136.

```
NFLG            --  addr
```
A variable used by -NUMBER that indicates whether the last conversion attempted was successful. NFLG is true if the conversion was successful; otherwise, it is false.

```
UMOVE           addr1  addr2  n  --
```
UMOVE is a "universal" memory move. It takes the block of memory n bytes long at addr1 and copies it to memory location addr2. UMOVE correctly uses either CMOVE or <CMOVE so that when a block of memory is moved onto part of itself, no data are destroyed.

```
"  cccc"     --          (at compile time)
   cccc:     -- addr     (at run time)
```
If compiling, the sequence cccc (delimited by the trailing ") is compiled into the dictionary as a string:

```
| len | c | c | c |...|  c |
```

All valFORTH strings are represented in this fashion. Since a single byte is used to store the length, a maximum string length of 255 is allowed. A string with 0 length is called a "null" string. At execution time, " puts the address in memory where the string is located onto the stack.

Note that " is IMMEDIATE. When executed outside of a colon definition, the string is not compiled into the dictionary, but is stored at PAD instead.

Example: " This is a string"

```
$CONSTANT cccc     $ --        (at compile time)
          cccc:      -- $      (at execution time)
```
Takes the string on top of the stack and compiles it into the dictionary with the name cccc. When cccc is later executed, the address of the string is pushed onto the stack.

Example: " Ready? <Y/N> " $CONSTANT VERIFY

```
$VARIABLE cccc      n  --
          cccc:       -- $
```
Reserves space for a string of length n. When cccc is later executed, the address of the string is pushed onto the stack.

Example: 80 $VARIABLE TEXTLINE

```
$.              $  --
```
Takes the string on top of the stack and sends it to the current output device.

Example: " Hi there" $. <ret> Hi there

```
$!              $  addr  --
```
Takes the string at second on stack and stores it at the address on top of stack.

Example: " Store me!" TEXTLINE $!

```
$+              $1  $2  --  $3
```
Takes $2 and concatenates it with $1, leaving $3 at PAD.

Example:
```
" Santa " $CONSTANT 1ST
" Claus"  $CONSTANT LAST
1ST LAST $+
$. <ret> Santa Claus
```

```
LEFT$           $1  n  --  $2
```
Returns the leftmost "n" characters of $1 as $2. $2 is stored at PAD.

Example:
```
" They"  3  LEFT$  $.  <ret>  The
```

```
RIGHT$          $1  n  --  $2
```
Returns the rightmost "n" characters of $1 as $2. $2 is stored at PAD.

Example:
```
" mother"  5  RIGHT$  $.  <ret>  other
```

```
MID$            $1  n  u  -- $2
```
Returns $2 of length u starting with the nth character of $1. Recall that the first character of a string is numbered as one.

Example:
```
" Timeout"  3  2  MID$  $.  <ret>  me
```

```
LEN             $ -- len
```
Returns the length of the specified string.

```
ASC             $ -- c
```
Returns the ASCII value of the first character of the specified string.

```
$COMPARE        $1  $2  --  flag
```
Compares $1 with $2 and returns a status flag. The flag is a) positive if $1 is greater than $2 or is equal to $2, but longer, b) zero if the strings match and are the same length, and c) negative if $1 less than $2 or if they are equal and $1 is shorter than $2.

```
$=              $1  $2  --  flag
```
Compares two strings on top of the stack and returns a status flag. The flag is true if the strings match and are equal in length, otherwise it is false.

```
$<              $1  $2  --  flag
```
Compares two strings on top of the stack and returns a status flag. The flag is true if $1 is less than $2 or if $1 matches $2 but is shorter in length.

```
$>              $1  $2  --  flag
```
Compares two strings on top of the stack and returns a status flag. The flag is true if $1 is greater than $2 or if $1 matches $2 but is longer in length.

```
SAVE$           $1  --  $2
```
As most string operations leave resultant strings at PAD, the word SAVE$ is used to temporarily move strings to PAD+512 so that they can be manipulated without being altered in the process.

Example:
```
" Wash"  SAVE$  " ington"  $+
```

INSTR $1 $2 -- n

Searches $1 for first occurrence of $2. Returns the character position in $1 if a match is found; otherwise, zero is returned.
Example: " FDCBA" $CONSTANT GRADES
GRADES " A" INSTR 1- . <ret> 4

CHR$ c -- $

Takes the character "c" and makes it into a string of length one and stores it at PAD.

DVAL $ -- d

Takes numerical string $ and converts it to a double length number. The variable NFLG is true if the conversion is successful, otherwise it is false. See -NUMBER above.
Example: " 123" DVAL D. <ret> 123

VAL $ -- n

Takes the numerical string $ and converts it to a single length number. The variable NFLG is true if the conversion is successful, otherwise it is false. See -NUMBER above.

DSTR$ d -- $

Takes the double number d and converts it to its ASCII representation as $ at PAD.
Example: 123 DSTR$ $. <ret> 123

STR$ n -- $

Takes the single length number n and converts it to its ASCII representation as $ at PAD.

STRING$ n $1 -- $2

Creates $2 as n copies of the first character of $1.

#IN$ n -- $

#IN$ has three similar but different functions. If n is positive, it accepts a string of n or fewer characters from the terminal. If n is zero, it accepts up to 255 characters from the terminal. If n is negative, it returns only after accepting -n characters from the terminal. The resultant string is stored at PAD.

IN$ -- $

Accepts a string of up to 255 characters from the terminal.

$-TB $1 -- $2

Removes trailing blanks from $1 leaving new $2.

$XCHG $1 -- $2

Exchanges the contents of $1 with $2.

ARRAYS and their COUSINS

All of the words described below create structures that are accessed in the same way, i.e., by putting the index or indices on the stack and then typing the structure's name. The differences are in the ways the structures are created.

The concept of the array should be known from BASIC. While in fig-FORTH there is no standard way to implement arrays and similar structures, there does exist a general consensus about how this should be done.

The point on which there is the most divergence of opinion is whether the first element in an array should be referred to by the index 0 or 1. We select 0 for the first index since this gives much cleaner code and makes more sense than 1 after you get used to it. (We've worked with it both ways.)

ARRAY and CARRAY, and 2ARRAY and 2CARRAY

The size of an array, specified when it is defined, is the number of elements in the array. In other words, an array defined by

```
8  ARRAY BINGO
```

will have 8 elements numbered 0 - 7.

To access an element of an array, do

```
n  array-name
```

to get the address of the nth element on the stack. (You will not be told if the number n is not a legitimate index number for the array.) For example,

```
5 BINGO
```

will leave the address of element number 5 in BINGO on the stack. You can store to or fetch from this address as you require.

The word CARRAY defines a byte or character array. A c-array works the same as an array, except that you must use C@ and C! to manipulate single elements, rather than @ and !.

The words 2ARRAY and 2CARRAY each take two numbers during definition of a 2ARRAY or 2CARRAY, and 2ARRAYS and 2CARRAYS take two numbers to access an element. Note that when using a 2CARRAY named, say, CHESSBOARD, and a constant named ROOK, the two phrases

```
ROOK   4  6  CHESSBOARD C!
        and
ROOK   6  4  CHESSBOARD C!
```

don't do the same thing. Also note that the phrase

```
8  8  2CARRAY CHESSBOARD
```

defines a 2CARRAY of 8 x 8 = 64 elements, with both indices running from 0 to 7.

When an ARRAY or a CARRAY is defined, the initial values of the elements are undefined.

TABLE AND CTABLE

A cousin of ARRAY is TABLE. Example: The phrase

```
TABLE  THISLIST  14 , 18 , -34 , 16 ,
```

defines a table THISLIST of 4 elements. (The commas above are part of the code and must be included.) The number of elements does not have to be specified. The elements in THISLIST are accessed using the indices 0-3, the same as if it had been defined as an array. The word CTABLE works similarly, though using C, instead of , to compile in the numbers. Note that negatives won't be compiled in by a C, since in two's complement representation negative numbers always occupy the maximum number of bytes.

VECTOR and CVECTOR

The last array-type words in this package are CVECTOR and VECTOR. Vector is just another name for a list. These words are used when the elements of the array you want to create are on the stack, with the last element on top of the stack. You just put the number of elements on the stack and the VECTOR or CVECTOR, and the name you want to use. Example:

```
-3  8 127 899 -43    5  VECTOR    POSITIONS
```

creates an array named POSITIONS with 5 elements 0-4 with -3 in element 0 and -43 in element 4. CVECTOR works in a similar way.

EXAMPLES:

```
2   3   BINGO   !
```

Stores the value 2 into element 3 of array BINGO.

```
2   THISLIST   @
```

Will leave the value in element 2 of table THISLIST. According to the definition of THISLIST above, this value will be -34.

```
3  POSITION  @  .  <cr>  899
```

ARRAY WORD GLOSSARY

```
ARRAY  cccc,    n --      (compiling)
       cccc:    m -- addr (executing)
```
When compiling, creates an array named cccc with n 16-bit elements numbered 0 thru n-1. Initial values are undefined. When executing, takes an argument, m, off the stack and leaves the address of element m of the array.

```
CARRAY  cccc,   n --      (compiling)
        cccc:   m -- addr (executing)
```
When compiling, creates a c-array named cccc with n 8-bit elements numbered 0 thru n-1. Initial values are undefined. When executing, takes an argument, m, off the stack and leaves the address of element m of the c-array.

```
TABLE   cccc,     --      (compiling)
        cccc:   m -- addr (executing)
```
When compiling, creates a table named cccc but does not allot space. Elements are compiled in directly with , (comma). When executing, takes one argument, m off the stack and, assuming 16-bit elements, leaves the address of element m of the table.

```
CTABLE   cccc,    --      (compiling)
         cccc:  m -- addr (executing)
```
When compiling, creates a c-table named cccc but does not allot space. Elements are compiled in directly with C, (c-comma). When executing, takes one argument, m off the stack and, assuming 8-bit elements, leaves the address of element m of the c-table.

```
X!     n0 ... nN  count addr --
```
Stores count 16-bit words, n0 thru nN into memory starting at addr, with n0 going into addr. Pronounced "extended store."

```
XC!   b0 ... bN  count addr --
```
Stores count 8-bit words, b0 thru bN into memory starting at addr, with b0 going into addr. Pronounced "extended c-store."

```
VECTOR  cccc,   n0 ... nN  count --   (compiling)
        cccc:          m  -- addr    (executing)
```
When compiling, creates a vector named cccc with count 16-bit elements numbered 0-N. n0 is the initial value of element 0, nN is the initial value of element N, and so on. When executing, takes one argument, m, off the stack and leaves the address of element m on the stack.

```
CVECTOR  cccc,  b0 ... bN  count --     (compiling)
         cccc:               m -- addr  (executing)
```
When compiling, creates a c-vector named cccc with count 8-bit elements numbered 0-N. b0 is the initial value of element 0, bN is the initial value of element N, and so on. When executing, takes an argument, m, off the stack and leaves the address of element m on the stack.

CASE STRUCTURES

It often becomes necessary to make many tests upon a single number. Typically, this is accomplished by using a series of nested "DUP test IF" statements followed by a series of ENDIFs to terminate the IFs. This is arduous and very wasteful of memory. valFORTH contains four very powerful Pascal-type CASE statements which ease programming and conserve memory.

The CASE: structure

Format:

```
CASE:  wordname
       word0
       word1
       ...
       wordN  ;
```

The word CASE: creates words that expect a number from 0 to N on the stack. If the number is zero, word0 is executed; if the number is one, the word1 is executed; and so on. No error checks are made to ensure that the case number is a legal value.

Example:

```
: ZERO   ." Zero" ;
: ONE    ." One" ;
: TWO    ." Two" ;

CASE:  NUM
       ZERO
       ONE
       TWO  ;

0  NUM  <ret>  Zero
1  NUM  <ret>  One
2  NUM  <ret>  Two
```

Note that any other number (e.g. 3 NUM) will crash the system.

The CASE Structure

Format:

```
: wordname
  ...
  CASE
     word0
     word1
     ...
     wordN
( NOCASE  wordnone )          (optional)
  CASEND
  ... ;
```

The CASE...CASEND structure is always used within a colon definition. Like CASE: above, it requires a number from 0 and N. However, unlike CASE: above, boundary checks are made so that an illegal case will do nothing. If the optional NOCASE clause is included then wordnone is executed if an "out of bounds" number is used.

Examples:

I)
```
: ZERO     ." Zero"  ;
: ONE      ." One"   ;
: TWO      ." Two"   ;

: CHECKNUM              ( n -- )
  CASE
     ZERO
     ONE
     TWO
  CASEND  ;

  0  CHECKNUM  <ret>  Zero
  1  CHECKNUM  <ret>  One
999  CHECKNUM  <ret>  (nothing happens)
  2  CHECKNUM  <ret>  Two
```

```
II)     : GRADEA    ." A" ;
        : GRADEB    ." B" ;
        : GRADEC    ." C" ;
        : GRADED    ." D" ;
        : OTHER     ." Failed" ;

        DECIMAL
        :  GETGRADE           ( -- )
           KEY 65 -           (Convert A to 0, B to 1, etc)
           CASE
              GRADEA
              GRADEB
              GRADEC
              GRADED
           NOCASE  OTHER
           CASEND  ;

        GETGRADE <return and press A> A
        GETGRADE <return and press B> B
        GETGRADE <return and press F> Failed
        GETGRADE <return and press D> D
```

The SEL Structure

Format:

```
     :  wordname
        ...
        SEL                          (Select)
           n1  ->  word0
           n2  ->  word1
           ...
           nN   >  wordN
     (  NOSEL  wordnone  )           (optional)
        SELEND
        ... ;
```

The SEL...SELEND structure is used when the "selection" numbers (n1 etc.) are not sequential. This structure is somewhat slower than either CASE or CASE: , but is much more general. SEL is typically used in operations such as table driver menus where single keystroke commands are used. The valFORTH video editor uses the SEL structure to implement the many editing keystroke commands.

Example:

```
I)      :  NICKEL    ." nickel."    ;
        :  DIME      ." dime."      ;
        :  QUARTER   ." quarter."   ;
        :  4BITS     ." fifty cent piece."   ;
        :  SUSANB    ." dollar"     ;
        :  BAD$$$    ." wooden nickel."   ;

        :  MONEY-NAME                ( n -- )
           ." That is called a "
           SEL
              5  ->  NICKEL
             10  ->  DIME
             25  ->  QUARTER
             50  ->  4BITS
            100  ->  SUSANB
          NOSEL  BAD$$$          ( this line is optional )
          SELEND  ;

  5 MONEY-NAME <ret> That is called a nickel.
 33 MONEY-NAME <ret> That is called a wooden nickel.
 25 MONEY-NAME <ret> That is called a quarter.
```

The COND Structure

Format:

```
     :  wordname
        ...
        COND
           condition0  <<  words0  >>
           condition1  <<  words1  >>
           ...
           conditionN  <<  wordsn  >>
     (  NOCOND  wordsnone  )              (optional)
        CONDEND
        ... ;
```

Unlike the three previous CASE structures which test for equality, the COND structure bases its selection upon any true conditional test (e.g. if n > 0 then...) COND can also be used for range cases. The NOCOND clause is optional and is only executed if no other condition passes. Only the code of the first condition that passes will be executed.

Example:

```
: EXAM                          ( score -- grade )
  COND
     90  >=   <<  ." Grade of A"  4  >>
     80  >=   <<  ." Grade of B"  3  >>
     70  >=   <<  ." Grade of C"  2  >>
     60  >=   <<  ." Grade of D"  1  >>
  NOCOND  ." Not too good"  0
  CONDEND  ;
```

Note that neither << nor >> are needed (nor allowed) around the "NOCOND" case. Also note that more than one word can be executed between the << and >> .

(intentionally left blank)

The following words extend the set of double number words to be as nearly identical as possible to the set in the book Starting FORTH. The exceptions are DVARIABLE and DCONSTANT which conform to the FIG standard by expecting initial values on the stack.

All of the single number operations comparable to the double number operations below were machine coded; all of the words below (with the exception of DVARIABLE) have high-level run time code and so are considerably slower than their single number counterparts.

DOUBLE NUMBER EXTENSION GLOSSARY

DVARIABLE cccc d --
cccc: -- addr
At compile time, creates a double number variable cccc with the initial value d. At run time, cccc leaves the address of its value on the stack.

DCONSTANT cccc d --
cccc: -- d
At compile time, creates a double number constant cccc with the initial value d. At run time, cccc leaves the value d on the stack.

0. -- 0.
A double number constant equal to double number zero.

1. -- 1.
A double number constant equal to double number one.

D- d1 d2 -- d3
Leaves d1-d2=d3.

D0= d -- flag
If d is equal to 0. leaves true flag; otherwise, leaves false flag.

D= d1 d2 -- flag
If d1 equals d2, leaves true flag; otherwise, leaves false flag.

D0< d -- flag
If d is negative, leaves true flag; otherwise, leaves false flag.

D< d1 d2 -- flag
If d1 is less than d2, leaves true flag; otherwise, leaves false flag.

D> d1 d2 -- flag
If d1 is greater than d2, leaves true flag; otherwise, leaves false flag.

DMIN d1 d2 -- d3
Leaves the minimum of d1 and d2.

DMAX d1 d2 -- d3
Leaves the maximum of d1 and d2.

D>R   d --
Sends the double number at top of stack to the return stack.

DR>   -- d
Pulls the double number at top of the return stack to the stack.

D,   d --
Compiles the double number at top of stack into the dictionary.

DU<   ud1 ud2 -- flag
If the unsigned double number ud1 is less than the unsigned double number ud2, leaves a true flag; otherwise, leaves a false flag.

M+   d1 n -- d2
Converts n to a double number and then sums with d1.

HIGH RESOLUTION TEXT OUTPUT

Occasionally, the need arises to print text in high resolution graphic displays (8 GR.). The following set of words explains how Graphic Characters can be used in valFORTH programs. The Graphic-Character output routines are designed to function identically to the standard FORTH output operations. There is an invisible cursor on the high resolution page which always points to where the next graphic-character will be printed. As with normal text output, this cursor can be repositioned at any time and in various ways. Because of the nature of hi-res printing, this cursor can also be moved vertically by partial characters. This allows for super/subscripting, over-striking, and underlining. Multiple character fonts on the same line are also possible.

GCINIT --

Initializes the graphic character output routines. This must be executed prior to using any other hi-res output words.

GC. n --

Displays the single length number n at the current hi-res cursor location.

GC.R n1 n2 --

Displays the single length number n1 right-justified in a field n2 graphic characters wide. See .R .

GCD.R d n --

Displays the double length number d right-justified in a field n graphic characters wide. See D.R .

GCEMIT c --

Displays the text character c at the current hi-res cursor location. Three special characters are interpreted by GCEMIT . The up arrow (↑) forces text output into the superscript mode; the down arrow (↓) forces the text into the subscript mode; and the left arrow (←) performs a GCBKS command (described below). See OSTRIKE below; also see EMIT.

GCLEN addr n -- len

Scans the first n characters at addr and returns the number of characters that will actually be displayed on screen. This is typically used to find the true length of a string that contains any of the non-printing special characters described in GCEMIT above. Used principally to aid in centering text, etc.

GCR --

Repositions the hi-res cursor to the beginning of the next hi-res text line. See CR .

GCLS --

Clears the hi-res display and repositions the cursor in the upper lefthand corner.

GCSPACE --
Sends a space to the graphic character output routine. See SPACE .

GCSPACES n --
Sends n spaces to the graphic character output routine. See SPACES .

GCTYPE addr n --
Sends the first n characters at addr to the graphic character output routine. See TYPE .

GC" cccc" --
Sends the character string cccc (delimited by ") to the graphic character output routine. If in the execution mode, this action is taken immediately. If in the compile mode, the character string is compiled into the dictionary and printed out only when executed in the word that uses it. See ." .

GCBKS --
Moves the hi-res cursor back one character position for overstriking or underlining.

GCPOS horz vert --
Positions the hi-res cursor to the coordinates specified. Note that the upper lefthand corner is 0,0.

GC$. addr --
Sends the string found at addr and preceded by a count byte to the graphic character output routine. See $. .

SUPER --
Forces the graphic character output routine into the superscript mode (or out of the subscript mode). See VMI below. May be performed within a string by the ^ character.

SUB --
Forces the graphic character output routine into the subscript mode (or out of the superscript mode). See VMI below. May be performed within a string by the v character.

VMI n --
Each character is eight bytes tall. The VMI command sets the number of eighths of characters to scroll up or down when either a SUPER or SUB command is issued. Normally, 4 VMI is used to scroll 4/8 or half a character in either direction.

VMI# -- addr
A variable set by VMI.

OSTRIKE ON or OFF --
The GCEMIT command has two separate functions. If OSTRIKE (overstrike) option is OFF, the character output will replace the character at the current cursor position. This is the normal method of output. If the OSTRIKE option is ON, the new character is printed over top of the previous character giving the impression of an overstrike. This allows the user to underline text and create new characters: Example: To do underline, a value of, say, 2 should be used with VMI, and then the v character added in the string before the underline character.

GCBAS -- addr

A variable which contains the address of the character set displayed by GCEMIT. To change character sets, simply store the address of your new character set into this variable.

GCLFT -- addr

A variable which holds the column position of the left margin. Normally two, this can be changed to obtain a different display window.

GCRGT -- addr

A variable which holds the column position of the right margin. Normally 39, this can be changed to obtain a different display window.

(intentionally left blank)

This is a grab-bag of useful words. Here they are...

XR/W #secs addr blk flag --

"Extended read-write." The same as R/W except that XR/W accepts a sector count for multiple sector reads and writes. Starting at address addr and block blk, read (flag true) or write (flag false) #secs sectors from or to disk.

SMOVE org des count --

Move count screens from screen # org to screen # dest.

The primary disk rearranging word, also used for moving sequences of screens between disks. This is a smart routine that uses all memory available below the current GR.-generated display list, with prompts for verification and disk swap if desired. See valFORTH Editor 1.1 documentation for further details.

LOADS start count --

Loads count screens starting from screen # start. This word is used if you want to use words that are not chained together by --> 's. It will stop loading if a CONSOLE button is held down when the routine finishes loading its present screen.

THRU start finish -- start count

Converts two range numbers to a start-count format. Example:

```
120 130 THRU PLISTS
```

will print screens 120 thru 130.

SEC n --
Provides an n second delay. Uses a tuned do-loop.

MSEC n --
Provides an n millisecond delay. (approx)
Uses a tuned do-loop.

H->L n -- b
Moves the high byte of n to the low byte and zero's the high byte, creating b. Machine code.

L->H n1 -- n2
Moves the low byte of n1 to the high byte and zero's the low byte, creating n2. Machine code.

H/L n1 -- n1(hi) n1(lo)
Split top of stack into two stack items: New top of stack is low byte of old top of stack. New second on stack is old top of stack with low byte zeroed.
Example: HEX 1234 H/L .S <cr> 1200 0034

BIT b -- n
Creates a number n that has only its bth bit set. The bits are numbered 0-15, with zero the least significant. Machine code.

?BIT n b -- f
Leaves a true flag if the bth bit of n is set. Otherwise leaves a false flag.

TBIT n1 b -- n2
Toggles the bth bit of n1, making n2.

SBIT n1 b -- n2
Sets the bth bit of n1, making n2.

RBIT n1 b -- n2
Resets the bth bit of n1, making n2.

STICK n -- horiz vert
Reads the nth stick (0-3) and resolves the setting into horizontal and vertical parts, with values from -1 to +1. -1 -1 means up and to the left.

PADDLE n1 -- n2
Reads the n1th paddle (0-7) and returns its value n2. Machine code.

ATTRACT f --
If the flag is true, the attract mode is initiated. If the flag is false, the attract mode is terminated.

NXTATR --
If the system is in the attract mode, this command cycles to the next color setup in the attract sequence. Disturbs the timer looked at by 16TIME.

HLDATR --
If the system is in attract mode, zero's fast byte of the system timer so that attract won't cycle to next color setup for at least four seconds or until system timer is changed, say by NXTATR. Disturbs the timer looked at by 16TIME.

16TIME -- n
Returns a 16 bit timer reading from the system clock at locations 19 and 20, decimal. This clock is updated 60 times per second, with the fast byte in 20. Machine code, not fooled by carry.

8RND -- b
Leaves one random byte from the internal hardware. Machine code.

16RND -- n
Leaves one random word from the internal hardware. Machine code with 20 cycle extra delay for rerandomization.

CHOOSE u1 -- u2
Randomly choose an unsigned number u2 which is less than u1.

CSHUFL addr n --
Randomly rearrange n bytes in memory, starting at address addr.
Pronounced "c-shuffle."

SHUFL addr n --
Randomly rearrange n words in memory, starting at address addr. Pronounced "shuffle." SHUFL may also be used to shuffle items directly on the stack by doing SP@ n SHUFL.

H. n --
See DEBUG Glossary.

A. addr --
Print the ASCII character at addr, or if not printable, print a period. (Used by DUMP).

DUMP addr n --
Starting at addr, dump at least n bytes (even multiple of 8) as ASCII and hex. May be exited early by pressing a CONSOLE button.

BLKOP system use only

BXOR addr count b --
Starting at address addr, for count bytes, perform bit-wise exclusive or with byte b at each address. Useful for toggling an area of display memory to inverse video or a different color, and for other purposes. For instance, in 0 GR., do

```
DCX  88 @  280  128  BXOR
```

Then do Shift-Clear to clear the screen. Pronounced "block ex or."

BAND addr count b --
Starting at address addr, for count bytes, perform bit-wise AND with byte b at each address. Applications similar to BXOR.
Pronounced "block and."

BOR addr count b -
Starting at address addr, for count bytes, perform bit-wise or with byte b at each address. Applications similar to BXOR.
Pronounced "block or."

STRIG n -- flag
Reads the button of joystick n (0-3). Leaves a true flag if the button is pressed, a false flag if it isn't.

PTRIG n -- flag
Reads the button of paddle n (0-7). Leaves a true flag if the button is pressed, a false flag if it isn't.

(intentionally left blank)

One of the more annoying parts about common releases of FORTH concerns the FORTH machine code assemblers. On the positive side, FORTH-based assemblers can be extraordinarily smart and easy to use interactively, and can compile on the fly as you type, rather than in multiple-pass fashion. (The 6502 assembler provided with valFORTH is a good example of a smart, structured, FORTH-based assembler.) On the other hand, since the assembler loads into the dictonary one usually sacrifices between 3 and 4K of memory on a utility that is only a compilation aid, and is not used during execution. With the utility described below, however, you can use the assembler and then remove it from the dictionary when you're finished with it.

In the directory of the Utilities/Editor disk (screen 170) you will find a heading of Transients. Loading this screen brings in three words: TRANSIENT, PERMANENT, and DISPOSE, and a few variables. It also defines a new area of memory called the Transient area. This area is used to load utilities like the assembler, certain parts of case statements, and similar constructs, that have one characteristic in common: They have compile-time behavior only, and are not used at run-time. An example will help make clear the sequence of operations. You may recall that on the valFORTH disk, in order to load floating point words you needed the assembler. Let's make a disk that has floating point but no assembler:

* Boot your valFORTH disk. It can be the bare system, or your normal programing disk if it doesn't have the assembler already in it.

* Insert your Utilities/Editor disk, find the Transient section in the directory, and load it.

* Do MTB (EMPTY-BUFFERS) and swap in your valFORTH disk. (It is a VERY good idea to get into the habit of doing MTB before swapping disks.) Find the assembler in the directory, but before you load it, do TRANSIENT to cause it to be loaded into the transient dictionary area, in high memory. Now go ahead and load the assembler. When it is loaded, do PERMANENT so that the next entries will go into the permanent dictionary area, which is back where you started.

* Now find and load the floating point words.

* Finally, do DISPOSE to pinch off the links that tie the transient area (with the assembler in it) to the permanent dictionary, with the floating point words in it. Do a VLIST or two to prove it to yourself. (Note that there are about a half-dozen words in the assembler vocabulary in the kernel. These were in the dictionary on boot up and are not affected by DISPOSE.)

You can derive great benefit from the simple recipe above, and if you study the Transient code a bit, you may learn even more. We offer several comments:

* In the case of the above recipe, you didn't actually have to do PERMANENT and TRANSIENT because the assembler source code checks at the front to see if TRANSIENT exists, and does it if so. At the end if checks to see if PERMANENT exists, and does it if so. This conditional execution is accomplished with the valFORTH construct

```
'(      )(      )
```

which is described in valFORTH documentation. Take a look at the assembler source code to see how this is done.

* If you want to do assembly on more than one section of code, you needn't DISPOSE until you really finished with the assembler; or, if you have DISPOSED of the assembler, you can bring it back in later without harm, by the same method. You can also code high-level definitions, and then more assembly code, and so on, and only do DISPOSE when you were finished. Be sure to do DISPOSE before SAVE or AUTO, however, because either your system will crash or your SAVE'd or AUTO'd program won't work.

The situation is slightly different with "case" words, since if you bring them in more than once you'll get duplicate names on the run-time words like (SEL), (CASE) and CASE:, which uses extra space and defeats the purpose of Transients.

* If you use the Transient structures for otherpurposes, remember only to send code that is not used at run-time to the transient area. As an example of this distinction, look at the code for the "case" words on the valFORTH disk. Note that the '( )( ) construct is again used, but that some of the parts of the case constructs, for instance (SEL), stay in the permanent dictionary. That is because (SEL) actually ends up in the compiled code, while SEL does not.

* Look at the beginning of the code for the Transient structures, and notice that the Transient area has been set up 4000 bytes below the display list. (The byte just below the display list in normal modes is pointed to by memory location 741 decimal, courtesy of the Atari OS.) This is usually a good place if only the 0 Graphics mode is used. (8 GR., for example, will over-write this area, crashing the system.) After DISPOSE is executed, this area is freed for other purposes. If you want to use a different area for Transients, just substitute your address into the source code on the appropriate screen. Remember that you must leave enough room for whatever will go into the Transient dictionary, and that NOTHING else must write to the area until you have cleared it out with DISPOSE. (This includes SMOVE, DISKCOPY1, DISKCOPY2, etc.)

****** NOTE ***** NOTE ***** NOTE ***** NOTE ***** NOTE ******

In the above example, 4000 bytes have been set aside for the Transient area just below the 0 GR. display list. This amount of memory will generally hold the assembler and some case statement compiling words. REMEMBER that if you have relocated the buffers (see the section on Relocating Buffers) to this area as well, you will have a collision, and a crashed system in short order.

To cure this, simply locate the Transient area 2113 bytes lower in memory so that there will be no overlap.

****** NOTE ***** NOTE ***** NOTE ***** NOTE ***** NOTE ******

ACKNOWLEDGEMENT

Various implementations of the Transient concept have appeared. valFORTH adopts the names TRANSIENT, PERMANENT, and DISPOSE from a public domain article by Phillip Wasson which appeared in FORTH DIMENSIONS volume III no. 6. The Transient structure implemented in the article has been altered somewhat in the valFORTH implementation to allow DISPOSE to dispose of the entire Transient structure, including DISPOSE itself, thus rendering the final product perfectly clean.

FORTH DIMENSIONS is a publication available through FIG (address listed elsewhere) and can be a valuable source of information and ideas to the advanced FORTH programmer.

```
Screen:  36
  0 ( Transients:  setup                )
  1 BASE @ DCX
  2
  3 HERE
  4
  5
  6 741 @ 4000 - DP !
  7 ( SUGGESTED PLACEMENT OF TAREA )
  8
  9
 10 HERE CONSTANT TAREA
 11    0 VARIABLE TP
 12    1 VARIABLE TPFLAG
 13      VARIABLE OLDDP
 14
 15                                ==>
```

```
Screen:  37
  0 ( Xsients: TRANSIENT PERMANENT )
  1
  2 : TRANSIENT                 ( -- )
  3   TPFLAG @ NOT
  4   IF HERE OLDDP ! TP @ DP !
  5      1 TPFLAG !
  6   ENDIF ;
  7
  8 : PERMANENT                 ( -- )
  9   TPFLAG @
 10   IF HERE TP ! OLDDP @ DP !
 11      0 TPFLAG !
 12   ENDIF ;
 13
 14
 15                                -->
```

```
Screen:  38
  0 ( Transients:  DISPOSE              )
  1 : DISPOSE   PERMANENT
  2   CR ." Disposing..." VOC-LINK
  3   BEGIN DUP    0 53279 C!
  4    BEGIN @ DUP TAREA U<
  5    UNTIL DUP ROT ! DUP 0=
  6   UNTIL DROP VOC-LINK @
  7   BEGIN DUP 4 -
  8    BEGIN DUP    0 53279 C!
  9     BEGIN PFA LFA @ DUP TAREA U<
 10     UNTIL
 11      DUP ROT PFA LFA ! DUP 0=
 12    UNTIL DROP @ DUP 0=
 13   UNTIL DROP [COMPILE] FORTH
 14   DEFINITIONS ." Done" CR ;
 15   PERMANENT              BASE !
```

```
Screen:  39
  0
  1
  2
  3
  4
  5
  6
  7
  8
  9
 10
 11
 12
 13
 14
 15
```

```
Screen:  40
  0
  1
  2
  3
  4
  5
  6
  7
  8
  9
 10
 11
 12
 13
 14
 15
```

```
Screen:  41
  0
  1
  2
  3
  4
  5
  6
  7
  8
  9
 10
 11
 12
 13
 14
 15
```

```
Screen:  42
  0 ( Utils:  CARRAY  ARRAY          )
  1 BASE @ HEX
  2 : CARRAY          ( cccc, n -- )
  3   CREATE SMUDGE ( cccc: n -- a )
  4     ALLOT
  5   ;CODE CA C, CA C, 18 C,
  6   A5 C, W C,  69 C, 02 C, 95 C,
  7   00 C, 98 C, 65 C, W 1+ C,
  8   95 C, 01 C, 4C C,
  9   ' + ( CFA @ ) , C;
 10
 11 : ARRAY           ( cccc, n -- )
 12   CREATE SMUDGE ( cccc: n -- a )
 13     2* ALLOT
 14   ;CODE 16 C, 00 C, 36 C, 01 C,
 15    4C C, ' CARRAY 08 + , C;  ==>
```

```
Screen:  43
  0 ( Utils:  CTABLE  TABLE          )
  1
  2 : CTABLE          ( cccc, -- )
  3   CREATE SMUDGE ( cccc: n -- a )
  4   ;CODE
  5   4C C, ' CARRAY 08 + , C;
  6
  7 : TABLE           ( cccc, -- )
  8   CREATE SMUDGE ( cccc: n -- a )
  9   ;CODE
 10   4C C, ' ARRAY 0A + , C;
 11
 12
 13
 14
 15                                -->
```

```
Screen:  44
  0 ( Utils:  2CARRAY  2ARRAY        )
  1
  2 : 2CARRAY       ( cccc, n n -- )
  3   <BUILDS       ( cccc: n n -- a )
  4    SWAP DUP , * ALLOT
  5   DOES>
  6     DUP >R @ * + R> + 2+ ;
  7
  8 : 2ARRAY        ( cccc, n n -- )
  9   <BUILDS       ( cccc: n n -- a )
 10    SWAP DUP , * 2* ALLOT
 11   DOES>
 12     DUP >R @ * + 2* R> + 2+ ;
 13
 14
 15                                ==>
```

```
Screen:  45
  0 ( Utils:  XC!  X!                )
  1
  2 : XC!     ( n0...nm cnt addr -- )
  3   OVER 1- + >R 0
  4   DO J I - C!
  5   LOOP R> DROP ;
  6
  7 : X!      ( n0...nm cnt addr -- )
  8   OVER 1- 2* + >R 0
  9   DO J I 2* - !
 10   LOOP R> DROP ;
 11
 12 ( Caution: Remember limitation
 13 ( on stack size of 30 values
 14 ( because of OS conflict. )
 15                                -->
```

```
Screen:  46
  0 ( Utils:  CVECTOR  VECTOR        )
  1
  2 : CVECTOR         ( cccc, cnt -- )
  3   CREATE SMUDGE ( cccc: n -- a )
  4   HERE OVER ALLOT XC!
  5   ;CODE
  6   4C C, ' CARRAY 08 + , C;
  7
  8 : VECTOR          ( cccc, cnt -- )
  9   CREATE SMUDGE ( cccc: n -- a )
 10   HERE OVER 2* ALLOT X!
 11   ;CODE
 12   4C C, ' ARRAY  0A + , C;
 13
 14
 15                        BASE !
```

```
Screen:  47
  0
  1
  2
  3
  4
  5
  6
  7
  8
  9
 10
 11
 12
 13
 14
 15
```

```
Screen:  48
 0 ( Utils:  HIDCHR  NOKEY  CURSOR)
 1 BASE @ DCX
 2
 3 '( CASE )( 28 KLOAD )
 4
 5 : HIDCHR                    ( -- )
 6   65535 94 ! ;
 7
 8 : NOKEY                     ( -- )
 9   255 764 C! ;  )
10
11 : CURSOR                  ( f -- )
12   0= 752 C!
13   28 EMIT 29 EMIT ;
14
15                                ==>
```

```
Screen:  49
 0 ( Utils:  INKEY$                 )
 1 DCX
 2 : (INKEY$)                 ( c -- )
 3   702 C! NOKEY ;
 4
 5 : INKEY$                   ( -- c )
 6   764 C@
 7   COND
 8     252 = (( 128 (INKEY$) 0 ))
 9     191 ) (( 0 ))
10     188 = (( 0 ))
11     124 = ((  64 (INKEY$) 0 ))
12      60 = ((   0 (INKEY$) 0 ))
13      39 = (( 0 ))
14   NOCOND KEY
15   CONDEND ;                    -->
```

```
Screen:  50
 0 ( Utils:  -Y/N                   )
 1
 2 : -Y/N                     ( -- f )
 3   BEGIN KEY
 4     COND
 5       89 = (( 1 1 ))
 6       78 = (( 0 1 ))
 7     NOCOND
 8       0
 9     CONDEND
10   UNTIL ;
11
12
13
14
15                                ==>
```

```
Screen:  51
 0 ( Utils:  Y/N  -RETURN  RETURN )
 1
 2 : Y/N                       ( -- f )
 3   ."  (Y/N) " -Y/N DUP
 4   IF 89 ELSE 78 ENDIF
 5   EMIT SPACE ;
 6
 7 : -RETURN                   ( -- )
 8   BEGIN KEY 155 = UNTIL ;
 9
10 : RETURN                    ( -- )
11   ."  (RETURN) " -RETURN ;
12
13
14
15                          BASE !
```

```
Screen:  52
 0 ( Screen code conversion words )
 1
 2 BASE @ HEX
 3
 4 CODE )BSCD              ( a a n -- )
 5   A9 C, 03 C, 20 C, SETUP ,
 6   HERE  C4 C, C2 C, D0 C, 07 C,
 7   C6 C, C3 C, 10 C, 03 C, 4C C,
 8   NEXT ,        B1 C, C6 C, 48 C,
 9   29 C, 7F C, C9 C, 60 C, B0 C,
10   0D C, C9 C, 20 C, B0 C, 06 C,
11   18 C, 69 C, 40 C, 4C C, HERE
12 2 ALLOT 38 C, E9 C, 20 C, HERE
13 SWAP !  91 C, C4 C, 68 C, 29 C,
14
15                                ==>
```

```
Screen:  53
 0 ( Screen code conversion words )
 1
 2   80 C, 11 C, C4 C, 91 C, C4 C,
 3   C8 C, D0 C, D3 C, E6 C, C7 C,
 4   E6 C, C5 C, 4C C, ,         C;
 5
 6 CODE BSCD)              ( a a n -- )
 7   A9 C, 03 C, 20 C, SETUP ,
 8   HERE  C4 C, C2 C, D0 C, 07 C,
 9   C6 C, C3 C, 10 C, 03 C, 4C C,
10   NEXT ,        B1 C, C6 C, 48 C,
11   29 C, 7F C, C9 C, 60 C, B0 C,
12   0D C, C9 C, 40 C, B0 C, 06 C,
13   18 C, 69 C, 20 C, 4C C, HERE
14 2 ALLOT 38 C, E9 C, 40 C, HERE
15                                -->
```

```
Screen:  54
 0 ( Screen code conversion words )
 1
 2 SWAP !  91 C, C4 C, 68 C, 29 C,
 3   80 C, 11 C, C4 C, 91 C, C4 C,
 4   C8 C, D0 C, D3 C, E6 C, C7 C,
 5   E6 C, C5 C, 4C C, ,
 6
 7
 8 : )SCD  SP@ DUP 1 )BSCD ;
 9 : SCD)  SP@ DUP 1 BSCD) ;
10
11
12
13
14
15                        BASE !
```

```
Screen:  55
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15
```

```
Screen:  56
 0 ( Case Statements:  CASE       )
 1 BASE @ DCX
 2 '( PERMANENT PERMANENT )(   )
 3 : (CASE)
 4   R C@ MIN -1 MAX 2*
 5   R 3 + + @EX
 6   R C@ 2* 5 + R) + )R ;
 7 '( TRANSIENT TRANSIENT )(   )
 8 : CASE
 9   ?COMP COMPILE (CASE)
10   HERE 0 C,
11   COMPILE NOOP 6 ;     IMMEDIATE
12
13 : NOCASE
14   6 ?PAIRS 7 ;         IMMEDIATE
15                                ==)
```

```
Screen:  57
 0 ( Case statements:  CASE       )
 1
 2 : CASEND
 3   DUP 6 =
 4   IF
 5     DROP COMPILE NOOP
 6   ELSE
 7     7 ?PAIRS
 8   ENDIF
 9   HERE 2- @ OVER 1+ !
10   HERE OVER -
11   5 - 2/ SWAP C! ;       IMMEDIATE
12
13 '( PERMANENT PERMANENT )(   )
14
15                                --)
```

```
Screen:  58
 0 ( Case statements:  SEL        )
 1
 2 '( PERMANENT PERMANENT )(   )
 3 : (SEL)
 4   R 1+ DUP 2+ DUP R C@
 5   2* 2* + R) DROP DUP )R SWAP
 6   DO I @ 3 PICK =
 7      IF I 2+ SWAP DROP LEAVE
 8      ENDIF
 9   4 /LOOP SWAP DROP @EX ;
10
11 '( TRANSIENT TRANSIENT )(   )
12 : SEL    ?COMP
13   ?LOADING COMPILE (SEL) HERE
14   0 C, COMPILE NOOP [COMPILE] [
15   8 ; IMMEDIATE                ==)
```

```
Screen:  59
 0 ( Case statements:  SEL        )
 1
 2 : NOSEL
 3   8 ?PAIRS [COMPILE] ' CFA
 4   OVER 1+ ! 8 ;        IMMEDIATE
 5
 6 : -)
 7   SWAP 8 ?PAIRS , DUP C@ 1+
 8   OVER C! [COMPILE] '
 9   CFA , 8 ;            IMMEDIATE
10
11 : SELEND
12   8 ?PAIRS
13   DROP [COMPILE] ] ;  IMMEDIATE
14 '( PERMANENT PERMANENT )(   )
15                                --)
```

```
Screen:  60
 0 ( Case statements:  COND          )
 1 '( TRANSIENT TRANSIENT )(  )
 2 : COND
 3   0 COMPILE DUP ;          IMMEDIATE
 4
 5 : ((
 6   1+ [COMPILE] IF
 7   COMPILE DROP ;           IMMEDIATE
 8
 9 : ))
10   [COMPILE] ELSE COMPILE
11   DUP ROT ;                IMMEDIATE
12
13 : NOCOND
14   COMPILE 2DROP ;          IMMEDIATE
15 '( PERMANENT PERMANENT )(  ) ==>
```

```
Screen:  61
 0 ( Case statements:  COND          )
 1
 2 '( TRANSIENT TRANSIENT )(  )
 3
 4 : CONDEND
 5   0 DO
 6     [COMPILE] ENDIF
 7   LOOP ;                   IMMEDIATE
 8
 9 '( PERMANENT PERMANENT )(  )
10
11
12
13
14
15                                  -->
```

```
Screen:  62
 0 ( Case statements:  CASE:         )
 1
 2 : CASE:
 3   (BUILDS
 4     SMUDGE !CSP
 5     [COMPILE] ]
 6   DOES)
 7     SWAP 2* + @EX ;
 8
 9
10
11
12
13
14
15                          BASE !
```

```
Screen:  63
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15
```

```
Screen:  64
 0 ( ValFORTH Video editor   V1.1 )
 1
 2 BASE @ DCX
 3
 4 '( XC! )( 21 KLOAD )
 5 '( HIDCHR )( 24 KLOAD )
 6 '( )BSCD )( 26 KLOAD )
 7
 8
 9
10
11
12
13
14
15                                  ==>
```

```
Screen:  65
 0 ( ValFORTH Video editor   V1.1 )
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15                                  -->
```

```
Screen:  66
 0 ( ValFORTH Video editor   V1.1 )
 1
 2 VOCABULARY EDITOR IMMEDIATE
 3 EDITOR DEFINITIONS
 4
 5 0 VARIABLE XLOC    ( X coord.    )
 6 0 VARIABLE YLOC    ( Y coord.    )
 7 0 VARIABLE INSRT   ( insert on?  )
 8 0 VARIABLE LSTCHR  ( last key    )
 9 0 VARIABLE ?BUFSM  ( buf same?   )
10 0 VARIABLE ?PADSM  ( PAD same?   )
11 0 VARIABLE ?ESC    ( coded char?)
12 0 VARIABLE TBLK    ( top block   )
13
14
15                                ==>
```

```
Screen:  67
 0 ( ValFORTH Video editor   V1.1 )
 1
 2 0 VARIABLE LNFLG   ( oops flag   )
 3 4     ARRAY UPSTAT ( update map  )
 4      15 CONSTANT 15
 5      32 CONSTANT 32
 6     128 CONSTANT 128
 7 5 32 * CONSTANT BLEN
 8
 9 : LMOVE 32 CMOVE ;
10 : BOL 88 @ YLOC @ 1+ 32 * + ;
11 : SBL  88 @ 544 + ;
12 : PBL  PAD 544 +  ;
13 : PBLL PBL BLEN + 32 - ;
14 : !SCR 88 @ 32 + PAD 512 BSCD) ;
15                                -->
```

```
Screen:  68
 0 ( ValFORTH Video editor   V1.1 )
 1
 2 : CURLOC                    ( -- )
 3   BOL XLOC @ + ;     ( SCR ADDR )
 4
 5 : CSHOW                     ( -- )
 6   CURLOC DUP     ( GET SCR ADDR )
 7   C@ 128 OR      ( INVERSE CHAR )
 8   SWAP C! ;      ( STORE ON SCR )
 9
10 : CBLANK                    ( -- )
11   CURLOC DUP     ( GET SCR ADDR )
12   C@ 127 AND     ( STRIP MSB )
13   SWAP C! ;      ( STORE IT )
14
15                                ==>
```

```
Screen:  69
 0 ( ValFORTH Video editor   V1.1 )
 1
 2 : UPCUR                     ( -- )
 3   CBLANK YLOC @
 4   1 - DUP 0<
 5   IF DROP 15 ENDIF
 6   YLOC ! CSHOW ;
 7
 8
 9 : DNCUR                     ( -- )
10   CBLANK YLOC @
11   1 + DUP 15 >
12   IF DROP 0 ENDIF
13   YLOC ! CSHOW ;
14
15                                -->
```

```
Screen:  70
 0 ( ValFORTH Video editor   V1.1 )
 1
 2 : LFCUR                     ( -- )
 3   CBLANK XLOC @
 4   1 - DUP 0<        ( AT L-SIDE?)
 5   IF DROP 31 ENDIF  ( FIX IF SO )
 6   XLOC ! CSHOW ;
 7
 8 : RTCUR                     ( -- )
 9   CBLANK XLOC @
10   1+ DUP 31 >       ( AT R-SIDE?)
11   IF DROP 0 ENDIF   ( FIX IF SO )
12   XLOC ! CSHOW ;
13
14 : EDMRK
15   1 YLOC @ 4 / UPSTAT ! ;      ==>
```

```
Screen:  71
 0 ( ValFORTH Video editor   V1.1 )
 1
 2 : INTGL                     ( -- )
 3   INSRT @ 0=       ( TOGGLE THE )
 4   INSRT ! ;        ( INSRT FLAG )
 5
 6 : NXTLN                     ( -- )
 7   CBLANK 0 XLOC !
 8   CSHOW DNCUR ;
 9
10 : CLREOL                    ( -- )
11   CBLANK !SCR
12   1 LNFLG ! CURLOC      ( CLEAR  )
13   32 XLOC @ -           ( TO END )
14   ERASE CSHOW           ( OF LINE)
15   EDMRK ;                      -->
```

```
Screen:  72
  0 ( ValFORTH Video editor    V1.1 )
  1
  2 : HMCUR                      ( -- )
  3   CBLANK 0 XLOC !
  4   0 YLOC ! CSHOW ;
  5
  6 : BYTINS CBLANK              ( -- )
  7   XLOC @ 31 (        ( SPREAD LN )
  8   IF
  9     CURLOC DUP 1+    ( FROM, TO )
 10     31 XLOC @ -      ( # CHARS )
 11     (CMOVE           ( MOVE IT )
 12   ENDIF
 13   0 CURLOC C!        ( CLEAR OLD )
 14   CSHOW EDMRK ;      ( CHARACTER )
 15                                ==>
```

```
Screen:  73
  0 ( ValFORTH Video editor    V1.1 )
  1
  2 : BYTDEL                     ( -- )
  3   CBLANK             ( CLOSE LINE)
  4   XLOC @ 31 (
  5   IF
  6     CURLOC DUP       ( FROM ADDR )
  7     1+ SWAP          ( TO ADDR )
  8     31 XLOC @ -      ( # CHARS )
  9     CMOVE            ( MOVE IT )
 10   ENDIF
 11   0 CURLOC           ( BLANK OUT )
 12   31 XLOC @ - + C!   ( CHAR AT )
 13   CSHOW EDMRK ;      ( END OF LN )
 14
 15                                -->
```

```
Screen:  74
  0 ( ValFORTH Video editor    V1.1 )
  1
  2 : LNINS                      ( -- )
  3   CBLANK 2 LNFLG ! !SCR
  4   4 YLOC @ 4 /
  5   DO 1 I UPSTAT ! LOOP
  6   YLOC @ 15 (
  7   IF
  8     BOL DUP 32 +
  9     15 YLOC @ - 32 *
 10     (CMOVE
 11   ENDIF
 12   BOL 32 ERASE
 13   CSHOW EDMRK ;
 14
 15                                ==>
```

```
Screen:  75
  0 ( ValFORTH Video editor    V1.1 )
  1
  2 : LNDEL                      ( -- )
  3   CBLANK 3 LNFLG ! !SCR
  4   4 YLOC @ 4 /
  5   DO 1 I UPSTAT ! LOOP
  6   YLOC @ 15 (
  7   IF BOL                   ( FROM )
  8     DUP 32 + SWAP          (  TO  )
  9     15 YLOC @ - 32 *       ( # CH )
 10     CMOVE
 11   ENDIF
 12   BOL 15 YLOC @ -
 13   32 * + 32 ERASE
 14   CSHOW EDMRK ;
 15                                -->
```

```
Screen:  76
  0 ( ValFORTH Video editor    V1.1 )
  1
  2 : BFSHW                      ( -- )
  3   PBLL 128 -             ( F , T  )
  4   SBL 160 CMOVE ;        ( # MOVE )
  5
  6 : BFROT                      ( -- )
  7   PBL DUP
  8   BLEN + LMOVE
  9   PBL DUP 32 +
 10   SWAP BLEN 32 -
 11   CMOVE PBLL 32 +
 12   PBLL LMOVE
 13   BFSHW ;
 14
 15                                ==>
```

```
Screen:  77
  0 ( ValFORTH Video editor    V1.1 )
  1
  2 : (BFROT                     ( -- )
  3   PBLL DUP
  4   32 + LMOVE
  5   PBL DUP 32 +
  6   BLEN 32 - (CMOVE
  7   PBL DUP BLEN +
  8   SWAP LMOVE
  9   BFSHW ;
 10
 11 : BFCLR                     ( -- )
 12   PBLL 32 ERASE
 13   (BFROT ;
 14
 15                                -->
```

```
Screen:  78
  0 ( ValFORTH Video editor   V1.1 )
  1
  2 : BFCPY                       ( -- )
  3   CBLANK BFROT       ( BRING LN )
  4   BOL PBLL           ( DOWN TO )
  5   LMOVE BFSHW        ( BUFFER & )
  6   CSHOW ;            ( ROTATE )
  7
  8 : )BFNXT BFCPY NXTLN ;      ( -- )
  9
 10 : )BFLN BFCPY LNDEL ;       ( -- )
 11
 12 : BFLN)                      ( -- )
 13   LNINS PBLL       ( TAKE LINE)
 14   BOL LMOVE        ( UP FROM )
 15   CSHOW (BFROT ; ( BUFFER  ) ==)
```

```
Screen:  79
  0 ( ValFORTH Video editor   V1.1 )
  1
  2 : BFRPL                       ( -- )
  3   CBLANK
  4   !SCR 4 LNFLG !     ( TAKE LINE )
  5   PBLL BOL LMOVE     ( UP TO SCR )
  6   (BFROT CSHOW       ( & ROTATE  )
  7   EDMRK ;
  8
  9 : TAB                         ( -- )
 10   CBLANK XLOC @ DUP
 11   31 = IF DROP -1 ENDIF
 12   4 + 4 / 4 * DUP 30 )
 13   IF DROP 31 ENDIF
 14   XLOC ! CSHOW ;
 15                              -->
```

```
Screen:  80
  0 ( ValFORTH Video editor   V1.1 )
  1
  2 : RUB                         ( -- )
  3   XLOC @ 0= NOT   ( ON L-EDGE? )
  4   IF
  5     LFCUR          ( RUB IF NOT )
  6     0 CURLOC C!
  7     CSHOW EDMRK
  8   ENDIF
  9   INSRT @
 10   IF
 11     BYTDEL
 12   ENDIF ;
 13
 14
 15                              ==)
```

```
Screen:  81
  0 ( ValFORTH Video editor   V1.1 )
  1
  2 : ARROW                       ( -- )
  3   CBLANK
  4   88 @ 541 + DUP @
  5   COND
  6     3341 = (( 30 7453 ))
  7     7453 = (( 00 0000 ))
  8   NOCOND
  9     30 3341
 10   CONDEND
 11   3 PICK !
 12   SWAP 2+ C!
 13   1 3 UPSTAT !
 14   CSHOW ;
 15                              -->
```

```
Screen:  82
  0 ( ValFORTH Video editor   V1.1 )
  1
  2 : OOPS                        ( -- )
  3   LNFLG @
  4   IF
  5     CBLANK
  6     PAD 88 @ 32 + 512 )BSCD
  7     CSHOW
  8     0 LNFLG !
  9   ENDIF ;
 10
 11
 12
 13
 14
 15                              ==)
```

```
Screen:  83
  0 ( ValFORTH Video editor   V1.1 )
  1
  2 : SPLIT                       ( -- )
  3   YLOC @ 15 (>
  4   IF
  5     CBLANK
  6     LNINS
  7     BOL DUP 32 + SWAP
  8     XLOC @ CMOVE
  9     BOL 32 +
 10     XLOC @ ERASE
 11     CSHOW
 12   ENDIF ;
 13
 14
 15                              -->
```

```
Screen:  84
  0 ( ValFORTH Video editor    V1.1 )
  1
  2 : SCRSV                        ( -- )
  3   88 @ 32 + PAD 512 BSCD>
  4   4 0
  5   DO
  6     I UPSTAT @
  7     0 I UPSTAT !
  8     IF
  9       PAD 128 I * +
 10       TBLK @ I + BLOCK
 11       128 CMOVE UPDATE
 12     ENDIF
 13   LOOP
 14   0 INSRT !
 15   0 XLOC ! 0 YLOC ! ;          ==>
```

```
Screen:  85
  0 ( ValFORTH Video editor    V1.1 )
  1
  2 : SCRGT                        ( -- )
  3   4 0
  4   DO
  5     TBLK @
  6     I + BLOCK
  7     PAD 128 I * +
  8     128 CMOVE
  9   LOOP
 10   PAD 88 @ 32 +
 11   512 >BSCD ;
 12
 13
 14
 15                                -->
```

```
Screen:  86
  0 ( ValFORTH Video editor    V1.1 )
  1
  2 : NWSCR              ( -1/0/1 -- )
  3   CBLANK DUP
  4   IF SCRSV ENDIF 2* 2*
  5   TBLK @ + 0 MAX TBLK ! SCRGT
  6   TBLK @ 8 /MOD
  7   DUP <ROT SCR !
  8   IF 44 ELSE 53 ENDIF
  9   ?1K NOT
 10   IF
 11     44 = SWAP 2* + DUP SCR ! 0
 12   ENDIF
 13   88 @ 17 + C!
 14   0 84 C! 11 85 ! 1 752 C!
 15   . 2 SPACES CSHOW ;           ==>
```

```
Screen:  87
  0 ( ValFORTH Video editor    V1.1 )
  1
  2 : PRVSCR -1 NWSCR ;            ( -- )
  3
  4 : NXTSCR  1 NWSCR ;            ( -- )
  5
  6 : SPLCHR  1 ?ESC ! ;           ( -- )
  7
  8 : EXIT                         ( -- )
  9   HMCUR 19 LSTCHR ! ;
 10
 11 : EDTABT                       ( -- )
 12   0 UPSTAT 8 ERASE
 13   EXIT ;
 14
 15                                -->
```

```
Screen:  88
  0 ( ValFORTH Video editor    V1.1 )
  1
  2 : PTCHR                        ( -- )
  3   INSRT @ EDMRK
  4   IF BYTINS ENDIF
  5   LSTCHR @ 127 AND
  6   DUP LSTCHR !
  7   >SCD CURLOC C!
  8   RTCUR XLOC @ 0=
  9   IF DNCUR ENDIF
 10   0 ?ESC ! CSHOW ;
 11
 12 : CONTROL                   ( n -- )
 13   SEL 19 -> EXIT     17 -> EDTABT
 14       28 -> UPCUR    29 -> DNCUR
 15                                ==>
```

```
Screen:  89
  0 ( ValFORTH Video editor    V1.1 )
  1
  2       30 -> LFCUR    31 -> RTCUR
  3      126 -> RUB     127 -> TAB
  4        9 -> INTGL   155 -> NXTLN
  5      255 -> BYTINS  254 -> BYTDEL
  6      157 -> LNINS   156 -> LNDEL
  7       18 -> BFROT     2 -> <BFROT
  8        3 -> BFCLR    11 -> >BFNXT
  9       20 -> >BFLN     6 -> BFLN>
 10       16 -> PRVSCR   14 -> NXTSCR
 11       27 -> SPLCHR    8 -> CLREOL
 12        1 -> ARROW    21 -> BFRPL
 13       15 -> OOPS     10 -> SPLIT
 14   NOSEL PTCHR
 15   SELEND ;                     -->
```

```
Screen:  90
 0 ( ValFORTH Video editor   V1.1 )
 1
 2 : (V)                    ( TBLK -- )
 3   DECIMAL
 4   DUP BLOCK DROP TBLK !
 5   1 PFLAG ! 0 GR. 1 752 C! CLS
 6   1 559 C@ 252 AND OR 559 C!
 7   112 560 @ 6 + C!
 8   112 560 @ 23 + C!
 9   ." Screen #" 11 SPACES
10   ." #Bufs: " BLEN 32 / . HIDCHR
11   0 UPSTAT 8 ERASE 0 NWSCR
12   PAD ?PADSM @ OVER ?PADSM ! =
13   PBL @ ?BUFSM @ = AND NOT
14   IF PBL BLEN ERASE ENDIF
15                                  ==>
```

```
Screen:  91
 0 ( ValFORTH Video editor   V1.1 )
 1   BFSHW
 2   BEGIN
 3     INKEY$ DUP LSTCHR ! -DUP
 4     IF
 5       ?ESC @
 6       IF DROP PTCHR @ LSTCHR !
 7       ELSE CONTROL ENDIF
 8     ELSE
 9       INSRT @
10       IF
11         CBLANK CSHOW
12       ENDIF
13     ENDIF
14     LSTCHR @ 19 =
15   UNTIL                          -->
```

```
Screen:  92
 0 ( ValFORTH Video editor   V1.1 )
 1
 2   CBLANK SCRSV @ 767 C!
 3   2 560 @ 6 + C!
 4   2 560 @ 23 + C!
 5   PBL @ ?BUFSM !
 6   2 559 C@ 252 AND OR 559 C!
 7   0 LNFLG ! 0 752 C! CLS CR
 8   ." Last edit on screen # "
 9   SCR @ . CR CR 0 INSRT ! ;
10
11 FORTH DEFINITIONS
12
13 : V                          ( s -- )
14   1 MAX B/SCR *
15   EDITOR (V) ;                   ==>
```

```
Screen:  93
 0 ( ValFORTH Video editor   V1.1 )
 1
 2 : L                              ( -- )
 3   SCR @ DUP 1+
 4   B/SCR * SWAP B/SCR *
 5   EDITOR TBLK @ DUP <ROT
 6   (= <ROT ) AND
 7   IF
 8     EDITOR TBLK @
 9   ELSE
10     SCR @ B/SCR *
11   ENDIF
12   EDITOR (V) ;
13
14
15                                  -->
```

```
Screen:  94
 0 ( ValFORTH Video editor   V1.1 )
 1
 2 : CLEAR                      ( s -- )
 3   B/SCR * B/SCR 0+S
 4   DO
 5    FORTH I BLOCK
 6    B/BUF BLANKS UPDATE
 7   LOOP ;
 8
 9 : COPY                   ( s1 s2 -- )
10   B/SCR * OFFSET @ +
11   SWAP B/SCR * B/SCR 0+S
12   DO DUP FORTH I
13     BLOCK 2- !
14     1+ UPDATE
15   LOOP DROP ( FLUSH ) ;          ==>
```

```
Screen:  95
 0 ( ValFORTH Video editor   V1.1 )
 1
 2 : CLEARS                    ( s # -- )
 3   OVER )R 0+S
 4   2DUP CR
 5   ." Clear from SCR " . CR
 6   ."         thru SCR " 1 - . Y/N
 7   IF
 8     DO
 9       FORTH I CLEAR
10     LOOP
11   ELSE
12     2DROP
13   ENDIF
14   R) SCR ! FLUSH ;
15                                  -->
```

```
Screen:  96
  0 ( ValFORTH Video editor   V1.1 )
  1
  2 : WHERE EDITOR          ( n n --- )
  3   OVER OVER
  4   DUP 65532 AND
  5   SWAP OVER - 128 *
  6   ROT + 32 /MOD
  7   YLOC C!
  8   2- 0 MAX XLOC C!
  9   1 INSRT !
 10   EDITOR (V) ;
 11
 12 : #BUFS                    ( # -- )
 13   5 MAX 320 MIN 32 * EDITOR
 14   ' BLEN ! 0 ?PADSM ! ;
 15                                ==>
```

```
Screen:  97
  0 ( ValFORTH Video editor   V1.1 )
  1
  2 : (LOC)                     ( sys )
  3   BLK @ , IN @ C, ;
  4
  5 : LOCATOR                   ( f -- )
  6   IF
  7     [ ' (LOC) CFA ] LITERAL
  8   ELSE
  9     [ ' NOOP CFA ] LITERAL
 10   ENDIF
 11   ' CREATE ! ;
 12
 13
 14
 15                                -->
```

```
Screen:  98
  0 ( ValFORTH Video editor   V1.1 )
  1
  2 : LOCATE
  3   [COMPILE] ' DUP NFA 1- DUP
  4   2- @ DUP 1439 U< SWAP 0# AND
  5   IF
  6     SWAP DROP DUP C@
  7     SWAP 2- @ WHERE 2DROP
  8   ELSE
  9     CR ." Cannot locate"
 10     '( DCMPR DROP DCMPR
 11     )( 2DROP CR )
 12   ENDIF ;
 13
 14
 15                        BASE !
```

```
Screen:  99
  0
  1
  2
  3
  4
  5
  6
  7
  8
  9
 10
 11
 12
 13
 14
 15
```

```
Screen: 100
  0
  1
  2
  3
  4
  5
  6
  7
  8
  9
 10
 11
 12
 13
 14
 15
```

```
Screen: 101
  0
  1
  2
  3
  4
  5
  6
  7
  8
  9
 10
 11
 12
 13
 14
 15
```

```
Screen: 102
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15
```

```
Screen: 103
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15
```

```
Screen: 104
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15
```

```
Screen: 105
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15
```

```
Screen: 106
 0 ( Hi-resolution text printing  )
 1
 2 BASE @ DCX
 3
 4 '( )SCD )( 26 KLOAD )
 5 '( COND )( 28 KLOAD )
 6
 7 57344 VARIABLE GCBAS
 8     0 VARIABLE GCPTR
 9     2 VARIABLE GCLFT
10    39 VARIABLE GCRGT
11     0 VARIABLE GMOD
12     0 VARIABLE GCCOL
13     0 VARIABLE GCROW
14   120 VARIABLE VMI#
15                              ==>
```

```
Screen: 107
 0 ( Hi-res:  GCR                 )
 1
 2 : GCR                       ( -- )
 3   1 GCROW @ + DUP 20
 4   703 C@ MAX (
 5   IF GCROW !
 6   ELSE
 7     DROP 88 @ 320 0+S
 8     703 C@ 4 =
 9     IF 6400 ELSE 7680 ENDIF 2DUP
10     + 320 - )R CMOVE
11     R) 320 ERASE
12   ENDIF
13   GCROW @ 320 *
14   GCLFT @ DUP GCCOL !
15   + GCPTR ! ;                 -->
```

```
Screen: 108
  0 ( Hi-res:  [GCEMIT]                    )
  1
  2 : (GCEMIT)                      ( c -- )
  3   )SCD 8 * GCBAS @ +
  4   GCPTR @ 88 @ + 320 0+S
  5   DO
  6     DUP C@ GMOD C@
  7     IF I C@ OR ENDIF
  8     I C! 1+
  9   40 /LOOP
 10   DROP 1 GCPTR +!
 11   1 GCCOL @ + DUP GCRGT @ )
 12   IF DROP GCR
 13   ELSE GCCOL !
 14   ENDIF ;
 15                                    ==>
```

```
Screen: 109
  0 ( Hi-res:  GCBKS OSTRIKE GCINIT)
  1
  2 : GCBKS                         ( -- )
  3   GCCOL @ GCLFT @ )
  4   IF
  5     -1 GCCOL +!     ( backspace )
  6     -1 GCPTR +!
  7   ENDIF ;
  8
  9 : OSTRIKE                       ( f -- )
 10   GMOD ! ;           ( overstrike)
 11
 12 : GCINIT                        ( -- )
 13   0 GCROW ! GCLFT @ DUP
 14   GCCOL ! GCPTR ! ;
 15                                    -->
```

```
Screen: 110
  0 ( Hi-res:  GCPOS SUPER SUB      )
  1
  2 : GCPOS              ( col row -- )
  3   2DUP 320 * + GCPTR !
  4   GCROW ! GCCOL ! ;
  5
  6 : SUPER                         ( -- )
  7   VMI# @ MINUS GCPTR +! ;
  8
  9 : SUB                           ( -- )
 10   VMI# @ GCPTR +! ;
 11
 12
 13
 14
 15                                    ==>
```

```
Screen: 111
  0 ( Hi-res:  GCEMIT  GCTYPE       )
  1
  2 : GCEMIT                    ( chr -- )
  3   DUP
  4   COND
  5     28 = (( DROP SUPER ))
  6     29 = (( DROP SUB   ))
  7     30 = (( DROP GCBKS ))
  8   NOCOND (GCEMIT)
  9   CONDEND ;
 10
 11 : GCTYPE          ( adr count -- )
 12   0 MAX -DUP
 13   IF 0+S DO I C@ GCEMIT LOOP
 14   ELSE DROP
 15   ENDIF ;                          -->
```

```
Screen: 112
  0 ( Hi-res:  [GC"]  GC"           )
  1
  2 : (GC")                         ( -- )
  3   R COUNT DUP 1+ R) + )R
  4   GCTYPE ;
  5
  6
  7 : GC"                           ( -- )
  8   34 STATE @
  9   IF
 10     COMPILE (GC") WORD
 11     HERE C@ 1+ ALLOT
 12   ELSE
 13     WORD HERE COUNT GCTYPE
 14   ENDIF ; IMMEDIATE
 15                                    ==>
```

```
Screen: 113
  0 ( Hi-res:  GCSPACE[S]  GCD.R    )
  1
  2 : GCSPACE                       ( -- )
  3   BL GCEMIT ;
  4
  5 : GCSPACES                   ( n -- )
  6   0 MAX -DUP
  7   IF 0
  8    DO GCSPACE
  9    LOOP
 10   ENDIF ;
 11
 12 : GCD.R                   ( d n -- )
 13   )R SWAP OVER DABS
 14   <# #S SIGN #> R) OVER -
 15   GCSPACES GCTYPE ;                -->
```

```
Screen: 114
  0 ( Hi-res:  GC.R  GC.  GCLEN    )
  1
  2 : GC.R                    ( n n -- )
  3   )R S-)D R) GCD.R ;
  4
  5 : GC.                       ( n -- )
  6   0 GC.R GCSPACE ;
  7
  8 : GCLEN      ( adr cnt -- #chrs )
  9   0 (ROT 0+S
 10   DO I C@ 28 -
 11      CASE 0 0 0
 12      NOCASE 1
 13      CASEND +
 14   LOOP ;
 15                                ==)
```

```
Screen: 115
  0 ( Hi-res:  VMI  GC.$            )
  1
  2 : VMI                       ( n -- )
  3   40 * VMI# ! ;
  4
  5 : GC$.                    ( adr -- )
  6   COUNT GCTYPE ;
  7
  8 : GCLS                        ( -- )
  9   88 @
 10   703 C@ 4 =
 11   IF 6400 ELSE 7680 ENDIF
 12   ERASE
 13   GCRGT @ 0 GCPOS ;
 14
 15 GCINIT                  BASE !
```

```
Screen: 116
  0
  1
  2
  3
  4
  5
  6
  7
  8
  9
 10
 11
 12
 13
 14
 15
```

```
Screen: 117
  0
  1
  2
  3
  4
  5
  6
  7
  8
  9
 10
 11
 12
 13
 14
 15
```

```
Screen: 118
  0
  1
  2
  3
  4
  5
  6
  7
  8
  9
 10
 11
 12
 13
 14
 15
```

```
Screen: 119
  0
  1
  2
  3
  4
  5
  6
  7
  8
  9
 10
 11
 12
 13
 14
 15
```

```
Screen: 120
  0 ( Double:  DVAR DCON D- D>R DR>)
  1 BASE @ DCX
  2
  3 : DVARIABLE       ( cccc -- adr )
  4   VARIABLE , ;
  5
  6 : DCONSTANT         ( cccc -- d )
  7   <BUILDS , ,
  8   DOES> D@ ;
  9
 10 0. DCONSTANT 0.  1. DCONSTANT 1.
 11
 12 : D-                  ( d d -- d )
 13   DMINUS D+ ;
 14
 15                                ==>
```

```
Screen: 121
  0 ( Double:  D0= D= D0< D< D>     )
  1
  2 : D0=                   ( d -- f )
  3   OR 0= ;
  4
  5 : D=                   ( d d -- f )
  6   D- D0= ;
  7
  8 : D0<                   ( d -- f )
  9   SWAP DROP 0< ;
 10
 11 : D<                   ( d d -- f )
 12   D- D0< ;
 13
 14 : D>                   ( d d -- f )
 15   2SWAP D< ;                   -->
```

```
Screen: 122
  0 ( Double:  DMIN  DMAX           )
  1
  2 : DMIN                 ( d d -- d )
  3   2OVER 2OVER D>
  4   IF
  5     2SWAP
  6   ENDIF
  7   2DROP ;
  8
  9 : DMAX                 ( d d -- d )
 10   2OVER 2OVER D<
 11   IF
 12     2SWAP
 13   ENDIF
 14   2DROP ;
 15                                ==>
```

```
Screen: 123
  0 ( Double:  D>R  DR>  D,  M+     )
  1
  2 : D>R                        ( d -- )
  3   R> <ROT SWAP >R >R >R ;
  4
  5 : DR>                        ( -- d )
  6   R> R> R> SWAP ROT >R ;
  7
  8 : D,                         ( d -- )
  9   , , ;
 10
 11 : M+                    ( d n -- d )
 12   S->D D+ ;
 13
 14
 15                                -->
```

```
Screen: 124
  0 ( Double:  DU<                  )
  1
  2 : DU<
  3   DUP 4 PICK XOR 0<
  4   IF
  5     2DROP D0< NOT
  6   ELSE
  7     D- D0<
  8   ENDIF ;
  9
 10
 11
 12
 13
 14
 15                          BASE !
```

```
Screen: 125
  0
  1
  2
  3
  4
  5
  6
  7
  8
  9
 10
 11
 12
 13
 14
 15
```

Screen: 126
```
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15
```

Screen: 127
```
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15
```

Screen: 128
```
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15
```

Screen: 129
```
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15
```

Screen: 130
```
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15
```

Screen: 131
```
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15
```

```
Screen: 132
  0
  1
  2
  3
  4
  5
  6
  7
  8
  9
 10
 11
 12
 13
 14
 15
```

```
Screen: 133
  0
  1
  2
  3
  4
  5
  6
  7
  8
  9
 10
 11
 12
 13
 14
 15
```

```
Screen: 134
  0 ( Utils:  Initialization          )
  1
  2 BASE @ DCX
  3
  4    '( XC! )( 21 KLOAD )
  5    '( HIDCHR )( 24 KLOAD )
  6    '( )BSCD )( 26 KLOAD )
  7
  8
  9
 10
 11
 12
 13
 14
 15                                ==>
```

```
Screen: 135
  0 ( Utils:                          )
  1
  2
  3
  4
  5
  6
  7
  8
  9
 10
 11
 12
 13
 14
 15                                -->
```

```
Screen: 136
  0 ( Utils:  XR/W                    )
  1
  2 : XR/W      ( #secs a blk# f -- )
  3   4 PICK 0
  4    DO
  5     3 PICK I B/BUF * +
  6     3 PICK I + 3 PICK R/W
  7    LOOP
  8    2DROP 2DROP ;
  9
 10
 11
 12
 13
 14
 15                                ==>
```

```
Screen: 137
  0 ( Utils:  SMOVE                   )
  1
  2 : SMOVE        ( org des cnt -- )
  3   FLUSH MTB
  4   741 @ PAD DUP 1 AND - - 2DUP
  5   SWAP B/SCR * B/BUF * U<
  6   IF CR ." Too many: "
  7    B/BUF B/SCR * / U.
  8    ." max." DROP 2DROP
  9   ELSE DROP
 10    )R DCX MTB CR
 11    ." SMOVE from " OVER DUP 3 .R
 12    ."  thru " R + 1- 3 .R CR
 13    8 SPACES
 14    ." to " DUP DUP 3 .R
 15    ."  thru " R + 1- 3 .R     -->
```

```
Screen: 138
 0 ( Utils:  SMOVE                    )
 1
 2    SPACE Y/N
 3    IF
 4     CR ." Insert source" RETURN
 5     R B/SCR * PAD DUP 1 AND -
 6     4 ROLL B/SCR * OFFSET @ +
 7     1 XR/W
 8     CR ." Insert dest." RETURN
 9     R> B/SCR * PAD DUP 1 AND -
10     ROT B/SCR * OFFSET @ +
11     0 XR/W
12    ELSE R> DROP 2DROP
13     CR ." Smove aborted..." CR
14    ENDIF
15   ENDIF ;                    ==>
```

```
Screen: 139
 0 ( Utils:  LOADS  THRU              )
 1
 2
 3 : LOADS              ( n cnt -- )
 4   0+S
 5   DO
 6    I LOAD ?EXIT
 7   LOOP ;
 8
 9
10 : THRU           ( n n -- n cnt )
11   OVER - 1+ ;
12
13
14
15                                 -->
```

```
Screen: 140
 0 ( Utils:  SEC  MSEC                )
 1
 2 : SEC                       ( n -- )
 3   0 DO
 4    9300 0
 5    DO
 6    LOOP
 7   LOOP ;
 8
 9 : MSEC                      ( n -- )
10   0 DO
11    6 0
12    DO
13    LOOP NOOP
14   LOOP ;
15                                 ==>
```

```
Screen: 141
 0 ( Utils:  H->L  L->H  H/L          )
 1
 2 HEX
 3
 4 CODE H->L                 ( n -- n )
 5       B5 C, 01 C, 95 C, 00 C,
 6       94 C, 01 C, 4C C, NEXT , C;
 7
 8 CODE L->H                 ( n -- n )
 9       B5 C, 00 C, 95 C, 01 C,
10       94 C, 00 C, 4C C, NEXT , C;
11
12 CODE H/L                 ( n -- n n )
13       B5 C, 00 C, 94 C, 00 C,
14       4C C, PUSH0A , C;
15 DCX                             -->
```

```
Screen: 142
 0 ( Utils:  BIT  ?BIT  TBIT          )
 1 HEX
 2 CODE BIT                  ( b -- n )
 3  B4 C, 00 C, C8 C, A9 C, 00 C,
 4  95 C, 00 C, 95 C, 01 C, 38 C,
 5  36 C, 00 C, 36 C, 01 C, 18 C,
 6  88 C, D0 C, F8 C, 4C C, NEXT ,
 7 C;
 8 : ?BIT BIT AND 0# ; ( n b -- f )
 9
10 : TBIT BIT XOR ;      ( n b -- n )
11
12 : SBIT BIT OR ;       ( n b -- n )
13
14 : RBIT                ( n b -- n )
15   FFFF SWAP TBIT AND ;          ==>
```

```
Screen: 143
 0 ( Utils:  STICK                    )
 1 HEX
 2 HERE DUP 2DUP   0 , 1 , -1 , 0 ,
 3
 4 CODE STICK              ( n -- h v )
 5
 6   B4 C, 00 C, B9 C, 78 C, 02 C,
 7   48 C, CA C, CA C, 29 C, 03 C,
 8   0A C, A8 C, B9 C,      , 95 C,
 9   02 C, C8 C, B9 C,      , 95 C,
10   03 C, 68 C, 4A C, 4A C, 29 C,
11   03 C, 0A C, A8 C, B9 C,     ,
12   95 C, 00 C, C8 C, B9 C,     ,
13   95 C, 01 C, 4C C, ' SWAP ,
14
15 CURRENT @ CONTEXT !             -->
```

```
Screen: 144
 0 ( Utils:  STRIG  PADDLE          )
 1 HEX
 2
 3
 4 CODE PADDLE              ( n -- n )
 5   B4 C, 00 C,  B9 C, 270 ,
 6   4C C, PUT0A , C;
 7
 8 CODE STRIG               ( n -- f )
 9   B4 C, 00 C, B9 C, 284 ,
10   49 C, 01 C, 4C C, PUT0A , C;
11
12 CODE PTRIG               ( n -- f )
13   B4 C, 00 C, B9 C, 27C ,
14   49 C, 01 C, 4C C, PUT0A , C;
15                                ==>
```

```
Screen: 145
 0 ( Utils:  ATRACT  NXTATR         )
 1
 2 DCX
 3
 4 : ATTRACT                 ( f -- )
 5   IF 255 ELSE 0 ENDIF 77 C! ;
 6
 7 : NXTATR
 8   255 20 C! ;             ( -- )
 9 ( Changes user clock )
10
11 : HLDATR
12   0 20 C! ;               ( -- )
13 ( Changes user clock )
14
15                                -->
```

```
Screen: 146
 0 ( Utils:  16TIME                 )
 1 HEX
 2
 3 CODE 16TIME
 4   CA C, CA C,
 5   A5 C, 13 C,  95 C, 01 C,
 6   A5 C, 14 C,  95 C, 00 C,
 7   D0 C, 04 C,
 8   A5 C, 13 C,  95 C, 01 C,
 9   4C C, NEXT , C;
10
11
12
13
14
15                                ==>
```

```
Screen: 147
 0 ( Utils:  8RND  16RND  CHOOSE    )
 1 HEX
 2
 3 CODE 8RND                 ( -- b )
 4   AD C, D20A ,
 5   4C C, PUSH0A ,
 6 C;
 7
 8 CODE 16RND                ( -- n )
 9   AD C, D20A , 48 C, 68 C, 48 C,
10   68 C, 48 C, AD  C, D20A ,
11   4C C, PUSH ,  C;
12
13 : CHOOSE                 ( n -- n )
14   16RND U* SWAP DROP ;
15                                -->
```

```
Screen: 148
 0 ( Utils:  CSHUFL  SHUFL          )
 1 DCX
 2 : CSHUFL                  ( a n -- )
 3   1- 0 SWAP
 4   DO
 5     DUP I CHOOSE + OVER I +
 6     2DUP C@ SWAP C@
 7     ROT C! SWAP C!
 8   -1 +LOOP DROP ;
 9
10 : SHUFL                   ( a n -- )
11   1- 0 SWAP
12   DO DUP I CHOOSE 2* +
13     OVER I 2* +
14     2DUP @ SWAP @ ROT ! SWAP !
15   -1 +LOOP DROP ;               ==>
```

```
Screen: 149
 0 ( Utils:  H.  A.                 )
 1
 2 : A.                      ( a -- )
 3   C@ 127 AND
 4   DUP 32 < OVER
 5   124 > OR
 6   IF DROP 46 ENDIF
 7   SPEMIT ;
 8
 9 '( H. -->  )(  )
10
11 : H.                     ( d -- )
12   BASE @ HEX SWAP
13   0 <# # # # #> TYPE
14   BASE ! ;
15                                -->
```

```
Screen: 150
  0 ( Utils:  DUMP                          )
  1 DCX
  2
  3 : DUMP                      ( a n -- )
  4   0+S
  5   DO
  6    CR I H-)L H. I H.
  7    2 SPACES I 8 0+S 2DUP
  8    DO
  9     I C@ H. SPACE
 10    LOOP CR 7 SPACES
 11    DO
 12     I A. 2 SPACES
 13    LOOP ?EXIT
 14   8 /LOOP
 15   CR ;                                ==)
```

```
Screen: 151
  0 ( Utils:  BLKOP  -- system      )
  1 HEX
  2
  3 CODE BLKOP  ( adr cnt byte -- )
  4   A9 C, 03 C, 20 C, SETUP ,
  5   HERE        C4 C, C4 C, D0 C,
  6   07 C, C6 C, C5 C, 10 C, 03 C,
  7   4C C, NEXT ,      B1 C, C6 C,
  8   A5 C, C2 C, 91 C, C6 C, C8 C,
  9   D0 C, EC C, E6 C, C7 C, 4C C,
 10   , DCX
 11 C;
 12
 13
 14
 15                                   --)
```

```
Screen: 152
  0 ( Utils:  BXOR                          )
  1 HEX
  2 CODE BXOR    ( adr cnt byte -- )
  3   A9 C, 45 C,
  4   8D C, ' BLKOP 12 + ,
  5   4C C, ' BLKOP ,  C;
  6
  7 CODE BAND    ( adr cnt byte -- )
  8   A9 C, 25 C,
  9   8D C, ' BLKOP 12 + ,
 10   4C C, ' BLKOP ,  C;
 11
 12 CODE BOR     ( adr cnt byte -- )
 13   A9 C, 05 C,
 14   8D C, ' BLKOP 12 + ,
 15   4C C, ' BLKOP , C;  BASE !
```

```
Screen: 153
  0
  1
  2
  3
  4
  5
  6
  7
  8
  9
 10
 11
 12
 13
 14
 15
```

```
Screen: 154
  0
  1
  2
  3
  4
  5
  6
  7
  8
  9
 10
 11
 12
 13
 14
 15
```

```
Screen: 155
  0
  1
  2
  3
  4
  5
  6
  7
  8
  9
 10
 11
 12
 13
 14
 15
```

```
Screen: 156
 0 ( Strings:  -TEXT                  )
 1 BASE @ DCX
 2 : -TEXT                ( a u a -- )
 3   2DUP + SWAP
 4   DO
 5     DROP 1+
 6     DUP 1- C@
 7     I C@ - DUP
 8     IF
 9       DUP ABS
10       / LEAVE
11     ENDIF
12   LOOP
13   SWAP DROP DUP
14   IF 1 SWAP +- ENDIF ;
15                                ==>
```

```
Screen: 157
 0 ( Strings:  -NUMBER                )
 1
 2 0 VARIABLE NFLG
 3
 4 : -NUMBER              ( addr -- d )
 5   BEGIN DUP C@ BL = DUP + NOT
 6   UNTIL 0 NFLG ! 0 0 ROT DUP 1+
 7   C@ 45 = DUP >R + -1
 8   BEGIN DPL ! (NUMBER) DUP C@
 9   DUP BL <> SWAP 0# AND
10   WHILE DUP C@ 46 - NFLG !
11   0 REPEAT DROP R> IF DMINUS
12   ENDIF NFLG @
13   IF 2DROP 0 0 ENDIF
14   NFLG @ NOT NFLG ! ;
15                                -->
```

```
Screen: 158
 0 ( Strings:  UMOVE , $!             )
 1
 2
 3 FORTH DEFINITIONS
 4
 5 : UMOVE                ( a a n -- )
 6   <ROT OVER OVER U<
 7   IF
 8     ROT <CMOVE
 9   ELSE
10     ROT CMOVE
11   ENDIF ;
12
13 : $!
14   OVER C@ 1+ UMOVE ;
15                                ==>
```

```
Screen: 159
 0 ( Strings:  $CON , $VAR , ["]  )
 1
 2 : $CONSTANT            ( $ ccc -- )
 3   PAD 512 + SWAP OVER $!
 4   0 VARIABLE -2 ALLOT
 5   HERE $! HERE C@ 1+ ALLOT ;
 6
 7 : $VARIABLE          ( len ccc -- )
 8   0 VARIABLE
 9   1- ALLOT ;
10
11 : (")                      ( -- $ )
12   R DUP C@ 1+ R> + >R ;
13
14
15                                -->
```

```
Screen: 160
 0 ( Strings:  "                      )
 1
 2 : "
 3   34 ( Ascii quote )
 4   STATE @
 5   IF                      ( cccc" -- )
 6     COMPILE (") WORD
 7     HERE C@ 1+ ALLOT
 8   ELSE
 9     WORD HERE       ( cccc" -- $ )
10     PAD $! PAD
11   ENDIF ;
12
13   IMMEDIATE
14
15                                ==>
```

```
Screen: 161
 0 ( Strings:  $. , $XCHG             )
 1
 2 : $.                        ( $ -- )
 3   DUP C@ 0>
 4   IF
 5     COUNT TYPE
 6   ELSE
 7     DROP
 8   ENDIF ;
 9
10
11 : $XCHG                ( $1 $2 -- )
12   DUP PAD 256 + $!
13   OVER SWAP $!
14   PAD 256 + SWAP $! ;
15                                -->
```

```
Screen: 162
 0 ( Strings:  $+ , LEFT$          )
 1
 2 : $+                ( $1 $2 -- $ )
 3   SWAP PAD 256 +
 4   )R R $!
 5   DUP C@ SWAP 1+
 6   R C@ 1+ R +
 7   3 PICK UMOVE
 8   R C@ + 255 MIN
 9   R C! R) PAD $! PAD ;
10
11 : LEFT$              ( $ N -- $ )
12   SWAP PAD (ROT PAD $!
13   OVER C@ MIN
14   OVER C! ;
15                                ==)
```

```
Screen: 163
 0 ( Strings:  RIGHT$ , MID$       )
 1
 2 : RIGHT$              ( $ n -- $ )
 3   SWAP PAD (ROT PAD $!
 4   OVER (ROT OVER C@
 5   DUP 4 PICK +
 6   (ROT MIN DUP
 7   (ROT 1- -
 8   SWAP ROT OVER OVER
 9   C! 1+ SWAP CMOVE ;
10
11 : MID$       ( $ start len -- $ )
12   3 PICK C@ 1+ ROT -
13   0 MAX ROT SWAP
14   RIGHT$ SWAP OVER
15   C@ MIN OVER C! ;             --)
```

```
Screen: 164
 0 ( Strings:  LEN , ASC , $COMP   )
 1
 2 : LEN               ( $ -- length )
 3   C@ ;
 4
 5 : ASC                   ( $ -- c )
 6   1+ C@ ;
 7
 8 : $COMPARE          ( $1 $2 -- f )
 9   2DUP C@ SWAP C@ SWAP
10   2DUP MIN (ROT - )R
11   ROT 1+ (ROT SWAP 1+
12   -TEXT -DUP 0=
13   IF R) DUP IF 1 SWAP +- ENDIF
14   ELSE R) DROP ENDIF ;
15                                ==)
```

```
Screen: 165
 0 ( Strings:  $( , $= , $) , SV$ )
 1
 2 : $(                ( $1 $2 -- f )
 3   $COMPARE 0( ;
 4
 5 : $=                ( $1 $2 -- f )
 6   $COMPARE 0= ;
 7
 8 : $)                ( $1 $2 -- f )
 9   $COMPARE 0) ;
10
11 : SAVE$                  ( $ -- $ )
12   PAD 512 + SWAP
13   OVER $! ;
14
15                                --)
```

```
Screen: 166
 0 ( Strings:  INSTR               )
 1
 2 0 VARIABLE INCNT
 3
 4 : INSTR             ( $1 $2 -- n )
 5   0 INCNT ! 1+ SWAP DUP
 6   )R OVER 1- C@ )R 1+
 7   DUP 1- C@ R - 1+ 0 MAX
 8   OVER + SWAP R) (ROT
 9   DO
10     2DUP I -TEXT 0=
11     IF
12       I J - INCNT ! LEAVE
13     ENDIF
14   LOOP
15   2DROP R) DROP INCNT @ ;      ==)
```

```
Screen: 167
 0 ( Strings:  CHR$ , DVAL , VAL   )
 1
 2 : CHR$                  ( c -- $ )
 3   1 PAD C!
 4   PAD 1+ C!
 5   PAD ;
 6
 7 : DVAL                  ( $ -- d )
 8   PAD $! PAD
 9   DUP C@ OVER 1+ +
10   0 SWAP C!
11   -NUMBER ;
12
13 : VAL                  ( $ -- n )
14   DVAL DROP ;
15                                --)
```

```
Screen: 168
  0 ( Strings:  DSTR$ , STR[ING]$  )
  1
  2 : DSTR$                ( d -- $ )
  3   DUP <ROT DABS
  4   <# #S SIGN #>
  5   SWAP 1- DUP
  6   <ROT C! PAD $! PAD ;
  7
  8 : STR$                 ( d -- $ )
  9   S->D DSTR$ ;
 10
 11 : STRING$           ( n $ -- $ )
 12   1+ C@ OVER
 13   PAD C! PAD
 14   1+ <ROT FILL PAD ;
 15                              ==>
```

```
Screen: 169
  0 ( Strings:  $-TB , #IN$ , IN$  )
  1
  2 : $-TB                 ( $ -- $ )
  3   DUP DUP 1+ SWAP C@
  4   -TRAILING SWAP DROP
  5   OVER C! ;
  6
  7 : #IN$                 ( n -- $ )
  8   -DUP 0= IF 255 ENDIF
  9   PAD 1+ SWAP EXPECT PAD
 10   BEGIN 1+ DUP C@ 0= UNTIL
 11   PAD 1+ - PAD C! PAD ;
 12
 13 : IN$                   ( -- $ )
 14   0 #IN$ ;
 15                       BASE !
```

```
Screen: 170
  0 CONTENTS OF THIS DISK:
  1
  2 TRANSIENTS:                36 LOAD
  3 ARRAYS & THEIR COUSINS:    42 LOAD
  4 KEYSTROKE WORDS:           48 LOAD
  5 SCREEN CODE CONVERSION:    52 LOAD
  6 CASE STATEMENTS:           56 LOAD
  7 valFORTH EDITOR 1.1:       64 LOAD
  8 HIGH-RES TEXT:            106 LOAD
  9 DOUBLE NUMBER XTNSIONS:   120 LOAD
 10 MISCELLANEOUS UTILS:      134 LOAD
 11 STRING WORDS:             156 LOAD
 12
 13
 14
 15
```

```
Screen: 171
  0
  1
  2
  3
  4
  5
  6
  7
  8
  9
 10
 11
 12
 13
 14
 15
```

```
Screen: 172
  0
  1
  2
  3
  4
  5
  6
  7
  8
  9
 10
 11
 12
 13
 14
 15
```

```
Screen: 173
  0
  1
  2
  3
  4
  5
  6
  7
  8
  9
 10
 11
 12
 13
 14
 15
```

```
Screen: 174
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15
```

```
Screen: 175
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15
```

```
Screen: 176
 0 ( Error messages                )
 1
 2 Stack empty
 3
 4 Dictionary full
 5
 6 Wrong addressing mode
 7
 8 Is not unique
 9
10 Value error
11
12 Disk address error
13
14 Stack full
15
```

```
Screen: 177
 0 Disk Error!
 1
 2 Dictionary too big
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15
```

```
Screen: 178
 0 ( Error messages                )
 1
 2 Use only in Definitions
 3
 4 Execution only
 5
 6 Conditionals not paired
 7
 8 Definition not finished
 9
10 In protected dictionary
11
12 Use only when loading
13
14 Off current screen
15
```

```
Screen: 179
 0 Declare VOCABULARY
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15
```

Placement of Tabs for valFORTH 1.1 Documentation

The tab titles included should be cut apart and inserted into the tabs in the following order, starting at the highest position:

* fig EDITOR Locate before section II

* 1.1 EXTENSIONS Locate before section IV

* 1.1 GLOSSARY Locate before section V

* ASSEMBLER Locate before section VI

April, 1982

Notes on Starting FORTH for the fig-Forth User

A very popular book on the FORTH language called Starting FORTH has recently been published. The author, Leo Brodie, gives an excellent description of the FORTH language as implemented at FORTH, Inc. fig-FORTH differs from that implementation in some areas, and this document explains those differences. All comments that apply to fig-FORTH also apply to valForth.

BLANK = BLANKS (page 285)

Brodie describes the word BLANK. In fig-FORTH, this word is BLANKS.

EMPTY-BUFFERS vs. EMPTY-BUFFERS (page 283)

Brodie's word EMPTY-BUFFERS does not necessarily change the buffers. In fig-FORTH, EMPTY-BUFFERS zero fills the buffers.

CONTEXT vs. CONTEXT (page 247)

These two words are not synonymous in the two versions. fig-FORTH uses a system of VOC-LINKS with CONTEXT, while FORTH, Inc. does not.

EXIT = ;S (page 246)

The word EXIT, as Brodie describes it, is identical in function to ;S in fig-FORTH.

'S = SP@ (page 247)

The word 'S in FORTH, Inc.'s is SP@ in fig-FORTH.

EMPTY (page 84)

Not yet implemented in fig-FORTH.

WIPE vs. CLEAR (page 84)

CLEAR requires a screen number while WIPE clears the last screen edited.

ABORT" (page 103)

Not implemented in fig-FORTH.

?DUP = -DUP (page 103)

The word ?DUP in FORTH, Inc.'s is -DUP in fig-FORTH.

?STACK vs. ?STACK (page 103)

?STACK as described by Brodie as incorrect for fig-FORTH. ?STACK in fig-FORTH automatically aborts if there is a stack error.

NEGATE = MINUS, DNEGATE = DMINUS (pages 123, 178)

The words NEGATE and DNEGATE in FORTH, Inc.'s are MINUS and DMINUS respectively in fig-FORTH.

+LOOP vs. +LOOP (page 143)

The word +LOOP, as Brodie describes it, works differently for negative stepping than the +LOOP in fig-FORTH. fig-FORTH always ends if the index equals the limit, even for negative stepping.

PAGE = CLS (page 143)

Brodie's PAGE is called CLS in valForth. It has no equivalent in fig-FORTH.

U/MOD = U/ (page 177)

Brodie's U/MOD is U/ in fig-FORTH.

CREATE vs. CREATE (page 209)

Brodie's CREATE works differently from CREATE in fig-FORTH. A word using CREATE in fig-FORTH must unSMUDGE the header before the word can be used. The ";" unsmudges headers automatically. In addition, Brodie's CREATE and fig-FORTH CREATE move different default values in the CFA of the created header (see below).

CREATE = <BUILDS (page 209)

In Brodie's chapter 11 on extending the compiler, he uses the series CREATE... DOES>. In fig-FORTH, this should be <BUILDS...DOES>.

NUMBER vs. NUMBER (page 285)

Brodie's NUMBER only converts numbers to double length if the double word set is loaded. fig-FORTH always converts numbers to double length.

>IN = IN, H = DP (page 247)

The variable >IN and H in Brodie's FORTH are IN and DP respectively in fig-FORTH.

VARIABLE vs. VARIABLE (page 209)

The word VARIABLE, as Brodie describes it, accepts no value from the stack. fig-FORTH, on the other hand, does expect an initialization value from the stack.

' vs. ' (page 215)

These words are not synonymous. ' in Brodie is the same as ' 2- in fig-FORTH (or, more properly, ' CFA).

INDEX 5/3/82

KEY TO INDEX ABBREVIATIONS

ABBR	MEANING	TAB	PACKAGE
$$	STRINGS	$-ARY-CASE-DBL	UTILITIES/EDITOR
ARY	ARRAYS	$-ARY-CASE-DBL	UTILITIES/EDITOR
CSE	CASE	$-ARY-CASE-DBL	UTILITIES/EDITOR
DBL	DBL# EXTENSIONS	$-ARY-CASE-DBL	UTILITIES/EDITOR
ASM	ASSEMBLER	ASSEMBLER	valFORTH 1.1
DBG	DEBUGGER	1.1 EXTENSIONS	valFORTH 1.1
FP	FLOATING POINT	1.1 EXTENSIONS	valFORTH 1.1
GCS	GRAF-COL-SOUND	1.1 EXTENSIONS	valFORTH 1.1
TOPD	TXT OUT, DISK PREP	1.1 EXTENSIONS	valFORTH 1.1
FE	fig EDITOR	fig EDITOR	valFORTH 1.1
VG1	valFORTH GLOSS	1.1 GLOSSARY	valFORTH 1.1
HRT	HI-RES TEXT	HRT-MSC-TRNS	UTILITIES/EDITOR
MSC	MISC. UTILITIES	HRT-MSC-TRNS	UTILITIES/EDITOR
TRNS	TRANSIENTS	HRT-MSC-TRNS	UTILITIES/EDITOR
VED1	valFORTH Ed. 1.1	valFORTH Ed. 1.1	UTILITIES/EDITOR

valFORTH SYSTEM INDEX

①

HANDY REFERENCE CARD

valFORTH 1.1 T.M

Stack inputs and outputs are shown; top of stack on right.
This card follows usage of the Forth Interest Group (S.F. Bay Area); usage aligned with the Forth 78 International Standard.

For more info: Forth Interest Group
P.O. Box 1105
San Carlos, CA 94070.

Operand Key:		
	n,n1,...	16-bit signed numbers
	d,d1,...	32-bit signed numbers
	u	16-bit unsigned number
	addr	address
	b	8-bit byte
	c	7-bit ascii character value
	f	boolean flag
	fp	floating point number
	s	string

Stack Manipulation

DUP	(n -- n n)	Duplicate top of stack.
DROP	(n --)	Throw away top of stack.
SWAP	(n1 n2 -- n2 n1)	Reverse top two stack items.
OVER	(n1 n2 -- n1 n2 n1)	Make copy of second item on top.
ROT	(n1 n2 n3 -- n2 n3 n1)	Rotate third item to top.
<ROT	(n1 n2 n3 -- n3 n1 n2)	Rotate top item to third.
?DUP	(n -- n ?)	Duplicate only if non-zero.
>R	(n --)	Move top item to "return stack" for temporary storage (use caution).
R>	(-- n)	Retrieve item from return stack.
R	(-- n)	Copy top of return stack onto stack.

Number Bases

DECIMAL	(--)	Set decimal base.
HEX	(--)	Set hexadecimal base.
BASE	(-- addr)	System variable containing number base.

Arithmetic and Logical

+	(n1 n2 -- sum)	Add.
D+	(d1 d2 -- sum)	Add double-precision numbers.
-	(n1 n2 -- diff)	Subtract (n1-n2).
*	(n1 n2 -- prod)	Multiply.
/	(n1 n2 -- quot)	Divide (n1/n2).
MOD	(n1 n2 -- rem)	Modulo (i.e. remainder from division).
/MOD	(n1 n2 -- rem quot)	Divide, giving remainder and quotient.
*/MOD	(n1 n2 n3 -- rem quot)	Multiply, then divide (n1*n2/n3), with double-precision intermediate.
*/	(n1 n2 n3 -- quot)	Like */MOD, but give quotient only.
MAX	(n1 n2 -- max)	Maximum.
MIN	(n1 n2 -- min)	Minimum.
ABS	(n -- absolute)	Absolute value.
DABS	(d -- absolute)	Absolute value of double-precision number.
MINUS	(n -- -n)	Change sign.
DMINUS	(d -- -d)	Change sign of double-precision number.
AND	(n1 n2 -- and)	Logical AND (bitwise).
OR	(n1 n2 -- or)	Logical OR (bitwise).
XOR	(n1 n2 -- xor)	Logical exclusive OR (bitwise).
NOT	(n -- f)	True if top number zero (i.e. reverses truth value)

Comparison

<	(n1 n2 -- f)	True if n1 less than n2.
>	(n1 n2 -- f)	True if n1 greater than n2.
<=	(n1 n2 -- f)	True if n1 less than or equal to n2.
>=	(n1 n2 -- f)	True if n1 greater than or equal to n2.
=	(n1 n2 -- f)	True if top two numbers are equal.
<>	(n1 n2 -- f)	True if n1 does not equal n2.
0<	(n -- f)	True if top number negative.
0>	(n -- f)	True if top number positive.
0=	(n -- f)	True if top number zero (i.e. reverses truth value.
0<>	(n -- f)	True if n does not equal zero.

Memory

@	(addr -- n)	Replace word address by contents.
!	(n addr --)	Store second word at address on top.
C@	(addr -- b)	Fetch one byte only.
C!	(b addr --)	Store one byte only.
?	(addr --)	Print contents of address.
C?	(addr --)	Print byte at address.
U?	(addr --)	Print unsigned contents of address.
+!	(n addr --)	Add second number on stack to contents of address on top.
CMOVE	(from to u --)	Move u bytes in memory from head to head.
<CMOVE	(from to u --)	Move u bytes in memory from tail to tail.
FILL	(addr u b --)	Fill u bytes in memory with b, beginning at address.
ERASE	(addr u --)	Fill u bytes in memory with zeroes, beginning at address.
BLANKS	(addr u --)	Fill u bytes in memory with blanks, beginning at address.

Control Structures

DO...LOOP	do: (end+1 start --)	Set up loop, given index range.
I	(-- index)	Place current index value on stack.
I'	(-- index)	Used to retrieve index after a >R.
J	(-- index)	Place index of outer DO-LOOP on stack.
LEAVE	(--)	Terminate loop at next LOOP, +LOOP, or /LOOP.
?EXIT	(--)	LEAVE if ?TERMINAL is true (i.e. pressed).
DO... +LOOP	do: (end+1 start --) +loop: (n --)	Like DO...LOOP, but adds stack value (instead of always '1') to index.
DO... /LOOP	do: (end+1 start --) /loop: (u --)	Like DO... +LOOP, but adds unsigned value to index.
IF...(true) ...ENDIF	if: (f --)	If top of stack true (non-zero), execute. (Note: Forth 78 uses IF...THEN.)
IF...(true) ...ELSE ...(false) ...ENDIF	if: (f --)	Same, but if false, execute ELSE clause. (Note: Forth 78 uses IF...ELSE...THEN.)
BEGIN... UNTIL	until: (f --)	Loop back to BEGIN until true at UNTIL. (Note: Forth 78 uses BEGIN...END.)
BEGIN... WHILE ...REPEAT	while: (f --)	Loop while true at WHILE,REPEAT loops unconditionally to BEGIN. (Note: Forth 78 uses BEGIN...IF ...AGAIN.)

Terminal Input - Output

.	(n --)	Print number.
.R	(n fieldwidth --)	Print number, right-justified in field.
D.	(d --)	Print double-precision number
D.R	(d fieldwidth --)	Print double-precision number, right-justified in field.
CR	(--)	Do a carriage return.
SPACE	(--)	Type one space.
SPACES	(n --)	Type n spaces.
."	(--)	Print message (terminated by ").
DUMP	(addr u --)	Dump u words starting at address.
TYPE	(addr u --)	Type string of u characters starting at address.
COUNT	(addr -- addr+1 u)	Change length-byte string to TYPE form.
?TERMINAL	(-- f)	True if terminal break request present.
KEY	(-- c)	Read key, put ascii value on stack.
EMIT	(c --)	Type ascii value from stack.
EXPECT	(addr n --)	Read n characters (or until carriage return) from input to address.
WORD	(c --)	Read one word from input stream, using given character (usually blank) as delimiter.

Input - Output Formating

NUMBER	(addr -- d)	Convert string at address to double-precision number.
<#	(--)	Start output string.
#	(d -- d)	Convert next digit of double-precision number and add character to output string.
#S	(d -- 0 0)	Convert all significant digits of double-precision number to output string.
SIGN	(n d -- d)	Insert sign of n into output string.
#>	(d -- addr u)	Terminate output string (ready for TYPE).
HOLD	(c --)	Insert ascii character into output string.

Disk Handling

LIST	(screen --)	List a disk screen.
LOAD	(screen --)	Load disk screen (compile or execute).
BLOCK	(block -- addr)	Read disk block to memory address.
B/BUF	(-- n)	System constant giving disk block size in bytes.
BLK	(-- addr)	System variable containing current block number.
SCR	(-- addr)	System variable containing current screen number.
UPDATE	(--)	Mark last buffer accessed as updated.
FLUSH	(--)	Write all updated buffers to disk.
EMPTY-BUFFERS	(--)	Erase all buffers.

Defining Words

: xxx	(--)	Begin colon definition of xxx.
;	(--)	End colon definition.
VARIABLE xxx	(n --) xxx: (-- addr)	Create a variable named xxx with initial value n; returns address when executed.
CONSTANT xxx	(n --) xxx: (-- n)	Create a constant named xxx with value n; returns value when executed.
CODE xxx	(--)	Begin definition of assembly-language primitive operative named xxx.
;CODE	(--)	Used to create a new defining word, with execution-time "code routine" for this data type in assembly.
<BUILDS... DOES>	does: (-- addr)	Used to create a new defining word, with execution-time routine for this data type in higher-level Forth.
LABEL xxx	(-- addr)	Creates a header xxx which when executed returns its PFA.

HANDY REFERENCE CARD

valFORTH 1.1 T.M.

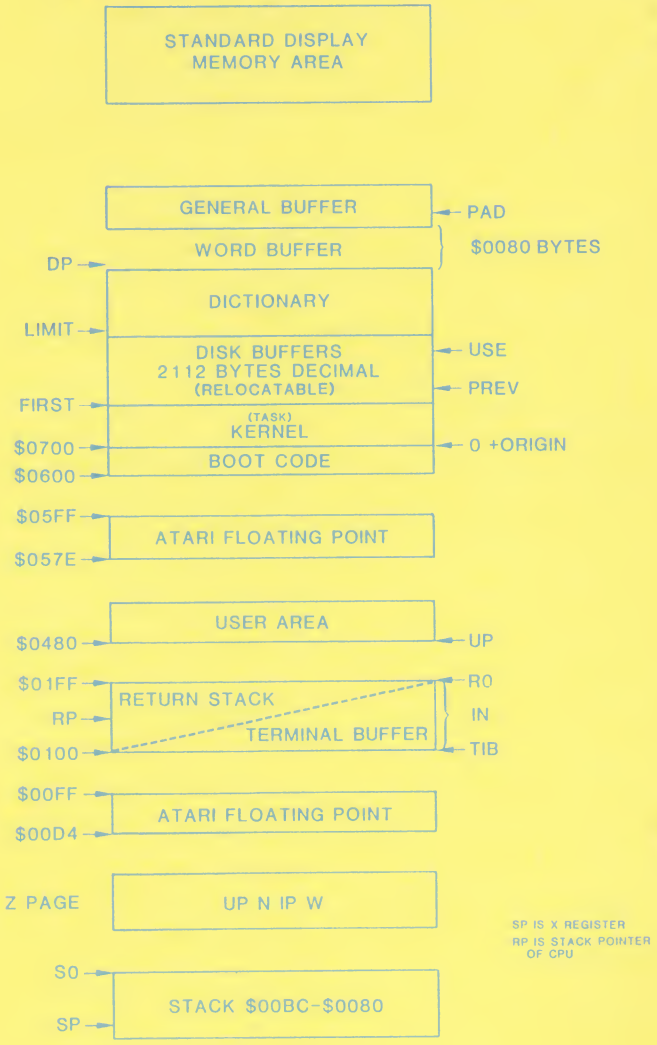
Vocabularies

CONTEXT	(-- addr)	Returns address of pointer to context vocabulary (searched first).
CURRENT	(-- addr)	Returns address of pointer to current vocabulary (where new definitions are put).
FORTH	(--)	Main Forth vocabulary (execution of FORTH sets CONTEXT vocabulary).
EDITOR	(--)	Editor vocabulary; sets CONTEXT.
ASSEMBLER	(--)	Assembler vocabulary; sets CONTEXT.
DEFINITIONS	(--)	Sets CURRENT vocabulary to CONTEXT.
VOCABULARY xxx	(--)	Create new vocabulary named xxx.
VLIST	(--)	Print names of all words in CONTEXT vocabulary.

Miscellaneous and System

(	--	Begin comment, terminated by right paren on same line; space after (.
FORGET xxx	--	Forget all definitions back to and including xxx.
ABORT	--	Error termination of operation.
'xxx	-- addr	Find the address of xxx in the dictionary; if used in definition, compile address.
HERE	-- addr	Returns address of next unused byte in the dictionary.
PAD	-- addr	Returns address of scratch area (usually 128 bytes beyond HERE).
IN	-- addr	System variable containing offset into input buffer. Used, e.g., by WORD.
SP@	-- addr	Returns address of top stack item.
ALLOT	n --	Leave a gap of n bytes in the dictionary.
,	n --	Compile a number into the dictionary.

valFORTH T.M. Memory Map



HANDY REFERENCE CARD

valFORTH 1.1

T.M.

Graphics and Color

SETCOLOR	(n1 n2 n3 --)	Color register n1 (0...3 and 4 for background) is set to hue n2 (0 to 15) and luminance n3 (0-14, even)
SE.	(n1 n2 n3 --)	Alias for SETCOLOR.
GR.	(n --)	Identical to GR. in BASIC. Adding 16 will suppress split display. Adding 32 will suppress display preclear. In addition, this GR. will not disturb player/missiles.
POS.	(x y --)	Same as BASIC POSITION or POS. Positions the invisible cursor if in a split display mode, and the text cursor if in 0 GR. .
POSIT	(x y --)	Positions and updates the cursor, similar to PLOT, but without changing display data.
PLOT	(x y --)	Same as BASIC PLOT. PLOTs point of color in register specified by last COLOR command, at point x y.
DRAWTO	(x y --)	Same as BASIC DRAWTO. Draws line from last PLOT'ted, DRAWTO'ed or POSIT'ed point to x y, using color in register specified by last COLOR command.
DR.	(x y --)	Alias for DRAWTO.
FIL	(b --)	Fills area between last PLOT'ted, DRAWTO'ed or POSIT'ed point to last position set by POS., using the color in register b.
G"	(--)	Used in the form G" ccccc". Sends text cccc to text area in non-0 Graphics mode, starting at current cursor position, in color of register specified by last COLOR command prior to cccc being output.
GTYPE	(addr count --)	Starting at addr, output count characters to text area in non-0 Graphics mode, starting at current cursor position, in color of register specified by last COLOR command.
LOC.	(x y -- b)	Positions the cursor at x y and fetches the data from display at that position. Like BASIC LOCATE and LOC. .
(G")	(--)	Run-time code compiled in by G".
POS@	(-- x y)	Leaves the x and y coordinates of the cursor on the stack.
CPUT	(b --)	Outputs the data b to the current cursor position.
CGET	(-- b)	Fetches the data b from the current cursor position
>SCD	(c1 -- c2)	Converts c1 from ATASCII to its display screen code, c2. Example: ASCII A >SCD 88 @ C! will put an "A" into the upper left corner of the display.
SCD>	(c1 -- c2)	Converts c1 from display screen code to ATASCII c2. See >SCD.
>BSCD	(addr1 addr2 count --)	Moves count bytes from addr1 to addr2, translating from ATASCII to display screen code on the way.
BSCD>	(addr1 addr2 count --)	Moves count bytes from addr1 to addr2, translating from display screen code to ATASCII on the way.
COLOR	(b --)	Saves the value b in the variable CLRBYT.
CLRBYT	(-- addr)	Variable that holds data from last COLOR command.

GREY	-- 0	PINK	-- 4	BLUE	-- 8	GREEN	-- 12
GOLD	-- 1	LVNDR	-- 5	LTBLUE	-- 9	YLWGRN	-- 13
ORNG	-- 2	BLPRPL	-- 6	TURQ	-- 10	ORNGRN	-- 14
RDORNG	-- 3	PRPLBL	-- 7	GRNBL	-- 11	LTORNG	-- 15

(CONSTANTs)

SOUND	(chan freq dist vol --)	Sets up the sound channel "chan" as indicated. Channel 0-3 Frequence: 0-255, 0 is highest pitch. Distortion: 0-14, evens only. Volume 0-15 Suggested mnemonic: CatFish Don't Vote
SO.	(chan freq dist vol --)	Alias of SOUND.
FILTER!	(n --)	Stores n in the audio control register and into the valFORTH shadow register, AUDCTL. Use AUDCTL when doing bit manipulation, then do FILTER!.
AUDCTL	(-- addr)	A variable containing the last value sent to the audio control register by FILTER!.
XSND	(n --)	Silences channel n
XSND4	(-)	Silences all channels.

Text Output and Disk Preparation

S:	(flag --)	If flag is true, enables handler that sends text to text screen. If false, disables the handler. (See PFLAG in main glossary.)
P:	(flag --)	If flag is true, enables handler that sends text to printer. If false, disables the handler. (See PFLAG in main glossary)
BEEP	(--)	Makes a raucous noise from the keyboard.
ASCII	(c, -- n (executing)) (c, -- (compiling))	Converts next character in input stream to ATASCII code. If executing, leaves on stack. If compiling, compiles as literal.
EJECT	(--)	Causes a form feed on smart printers if the printer handler has been enabled by ON P:. May need adjustment for dumb or nonstandard printers
LISTS	(start count --)	From start, lists count screens. May be aborted by CONSOLE button at the end of a screen.
PLIST	(scr --)	Lists screen scr to the printer, then restores former printer handler status.
PLISTS	(start cnt --)	From start, lists cnt screens to printer three to a page, then restores former printer handler status. May be aborted by CONSOLE button at the end of a screen.
FORMAT	(--)	With prompts, will format a disk in drive of your choice.

Debugging Utilities

DECOMP	xxx	Does a decompilation of the word xxx if it can be found in the active vocabularies.
CDUMP	(addr n --)	A character dump from addr for at least n characters. (Will always do a multiple of 16.)
#DUMP	(addr n --)	A numerical dump in the current base for at least n characters. (Will always do a multiple of 8.)
(FREE)	(-- n)	Leaves number of bytes between bottom of display list and PAD.
FREE	(--)	Does (FREE) and then prints the stack and "bytes".
H.	(n --)	Prints n in HEX, leaves BASE unchanged.
STACK	(flag --)	If flag is true, turns on visible stack. If flag is false, turns off visible stack.
.S	(... -- ...)	Does a signed, nondestructive stack printout, TOS at right. Also sets visible stack to do signed printout.
U.S	(... -- ...)	Does unsigned, nondestructive stack printout, TOS at right. Also sets visible stack to do unsigned printout.
B?	(--)	Prints the current base, in decimal. Leaves BASE undisturbed.
CFALIT	xxx (-- cfa (executing)) xxx (-- (compiling))	Gets the cfa (code field address) of xxx. If executing, leaves it on the stack; if compiling, compiles it as a literal.

Floating Point

FCONSTANT	xxx (fp --) xxx (-- fp)	The character string is assigned the constant value fp. When xxx is executed, fp will be put on the stack.
FVARIABLE	xxx (fp --) xxx: (addr --)	The character string xxx is assigned the initial value fp. When xxx is executed, the addr (two bytes) of the value of xxx will be put on the stack
FDUP	(fp1 -- fp1 fp1)	Copies the fp number at top-of-stack.
FDROP	(fp --)	Discards the fp number at top-of-stack.)
FOVER	(fp2 fp1 -- fp2 fp1 fp2)	Copies the fp number at 2nd-on-stack to top-of-stack.
FLOATING	xxx (-- fp)	Attempts to convert the following string, xxx, to a fp number.
FP	xxx (-- fp)	Alias for FLOATING.
F@	(addr -- fp)	Fetches the fp number whose address is at top-of-stack.
F!	(fp addr --)	Stores fp into addr. Remember that the operation will take six bytes in memory.
F.	(fp --)	Type out the fp number at top-of-stack. Ignores the current value in BASE and uses base 10.
F?	(addr --)	Fetches a fp number from addr and types it out.
F+	(fp2 fp1 -- fp3)	Replaces the two top-of-stack fp items, fp2 and fp1, with their fp sum, fp3.
F-	(fp2 fp1 -- fp3)	Replaces the two top-of-stack fp items fp2 and fp1, with their difference, fp3=fp2-fp1.
F*	(fp2 fp1 -- fp3)	Replaces the two top-of-stack fp items fp2 and fp1, with their product, fp3.
F/	(fp2 fp1 -- fp3)	Replaces the two top-of-stack fp items fp2 and fp1, with their quotient, fp3=fp2/fp1.
FLOAT	(n -- fp)	Replaces number at top-of-stack with its fp equivalent.
FIX	(fp (non-neg, less than 32767.5) -- n)	Replaces fp number at top-of-stack, constrained as indicated, with its integer equivalent.
LOG	(fp1 -- fp2)	Replaces fp1 with its base e logarithm, fp2. Not defined for fp1 negative.
LOG10	(fp1 -- fp2)	Replaces fp1 with its base 10 decimal logarithm, fp2. Not defined for fp1 negative.
EXP	(fp1 -- fp2)	Replaces fp1 with fp2, which equals e to the power fp1.
EXP10	(fp1 -- fp2)	Replaces fp1 with fp2, which equals 10 to the power fp1
F0=	(fp -- flag)	If fp is equal to floating-point 0, a true flag is left. Otherwise, a false flag is left.
F=	(fp2 fp1 -- flag)	If fp2 is equal to fp1, a true flag is left. Otherwise, a false flag is left.
F>	(fp2 fp1 -- flag)	If fp2 is greater than fp1, a true flag is left. Otherwise, a false flag is left.
F<	(fp2 fp1 -- flag)	If fp2 is less than fp1, a true flag is left. Otherwise, a false flag is left.
FLITERAL	(fp --)	If compiling, then compile the fp stack value as a fp literal.

Operating System

OPEN	(addr n0 n1 n2 -- n3)	This word opens the device whose name is at addr. The device is opened on channel n0 with AUX1 and AUX2 as n1 and n2 respectively. The device status byte is returned as n3.
CLOSE	(n --)	Closes channel n.
PUT	(b1 n -- b2)	Outputs byte b1 on channel n, returns status byte b2.
GET	(n -- b1 b2)	Gets byte b1 from channel n, returns status byte b2.
GETREC	(addr n1 n2 -- n3)	Inputs record from channel n2 up to length n1. Returns status byte n3
PUTREC	(addr n1 n2 -- n3)	Outputs n1 characters starting at addr through channel n2. Returns status byte n3.
STATUS	(n -- b)	Returns status byte b from channel n.
DEVSTAT	(n -- b1 b2 b3)	From channel n1 gets device status bytes b1 and b2, and normal status byte b3.
SPECIAL	(b1 b2 b3 b4 b5 b6 b7 b8 -- b9)	Implements the Operating System "Special" command. AUX1 through AUX6 are b1 through b6 respectively, command byte is b7, channel number is b8. Returns status byte b9.
RS232	(--)	Loads the Atari 850 drivers into the dictionary (approx 1.8K).

HANDY REFERENCE CARD

valFORTH 1.1™

valFORTH 6502 Assembler

ASSEMBLER	(---)	Calls up the assembler vocabulary for subsequent assembly language programming.
CODE xxx	(---)	Enters the new word "xxx" into the dictionary as machine language word and calls up the assembler vocabulary for subsequent assembly language programming.
C;	(---)	Terminates an assembly language definition by performing a security check and setting the CONTEXT vocabulary to the same as the CURRENT vocabulary.
END-CODE	(---)	A commonly used synonym for the word C; above. The word C; is recommended over END-CODE.
SUBROUTINE	xxx (---)	Enters the new word 'xxx" into the dictionary as machine language subroutine and calls up the assembler vocabulary for subsequent assembly language programming.
;CODE	(---)	When the assembler is loaded, puts the system into the assembler vocabulary for subsequent assembly language programming. See main glossary for further explanation.

Control Structures

IF,	(flag --- addr 2)	Begins a machine language control structure based on the 6502 status flag on top of the stack. Leaves an address and a security check value for the ELSE, or ENDIF, clauses below. 'flag' can be EQ , NE , CC , CS , VC , VS , MI , or PL . Command forms: ...flag..IF,..if-true..ENDIF,...all... ...flag..IF,..if-true.. ELSE,..if-false..ENDIF,..all...
ELSE,	(addr 2 --- addr 3)	Used in an IF, clause to allow for execution of code only if IF, clause is false. If the IF, clause is true, this code is bypassed.
ENDIF,	(addr 2/3 ---)	Used to terminate an IF, control structure clause. Additionally, ENDIF, resolves all forward references. See IF, above for command form.
BEGIN,	(--- addr 1)	Begins machine language control structures of the following forms: ...BEGIN,... AGAIN,... .. BEGIN,...flag .UNTIL,... ...BEGIN,. .flag..WHILE,..while-true .REPEAT,... where "flag" is one of the 6502 statuses: EQ , NE , CC , CS , VC , VS , MI , and PL .
UNTIL,	(addr 1 flag ---)	Used to terminate a post-testing BEGIN, clause thus allowing for conditional looping of a program segment while "flag" is false.
WHILE,	(addr 1 flag --- addr 4)	Used to begin a pre-testing BEGIN, clause thus allowing for conditional looping of a program segment while "flag" is true.
REPEAT,	(addr 4 ---)	Used to terminate a pre-testing BEGIN,..WHILE, clause. Additionally, REPEAT, resolves all forward addresses of the current WHILE, clause.
AGAIN,	(addr 1 ---)	Used to terminate an unconditional BEGIN, clause. Execution cannot exit this loop unless a JMP, instruction is used.

Parameter Passing (These routines must be jumped to.)

NEXT	(--- addr)	Transfers control to the next FORTH word to be executed. The parameter stack is left unchanged.
PUSH	(--- addr)	Pushes a 16 bit value to the parameter stack whose low byte is found on the 6502 return stack and whose high byte is found in the accumulator.
PUSHOA	(--- addr)	Pushes a 16 bit value to the parameter stack whose low byte is found in the accumulator and whose high byte is zero.
PUT	(--- addr)	Replaces the value currently on top of the parameter stack with the 16 bit value whose low byte is found on the 6502 stack and whose high byte is in the accumulator.
PUTOA	(--- addr)	Replaces the value currently on top of the parameter stack with the 16 bit value whose low byte is in the accumulator and whose high byte is set to zero.
BINARY	(--- addr)	Drops the top value of the parameter stack and then performs a PUT operation described above.
POP and POPTWO	(--- addr)	POP drops one value from the parameter stack. POPTWO drops two values from the parameter stack
SETUP	(--- addr)	Moves one to four values to the N scratch area in the zero page and drops all values moved from the parameter stack.
N	(--- addr)	Points to a nine-byte scratch area in the zero page beginning at N-1 and going to N+7.
Opcodes	(various --- various)	ADC, AND, ASL, BIT, BRK, CLC, CLD, CLI, CLV, CMP, CPX, CPY, DEC, DEX, DEY, EOR, INC, INX, INY, JSR, JMP, LDA, LDX, LDY, LSR, NOP, ORA, PHA, PHP, PLA, PLP, ROL, ROR, RTI, RTS, SBC, SEC, SED, SEI, STA, STX, TAX, TAY, TSX, TXA, TXS, TYA,

Aliases

NXT,	= NEXT JMP,	POP2,	= POPTWO JMP,
PSH,	= PUSH JMP,	XL,	= XSAVE LDX,
PUT,	= PUT JMP,	XS,	= XSAVE STX,
PSHA,	= PUSHOA JMP,	THEN,	= ENDIF,
PUTA,	= PUTOA JMP,	END,	= UNTIL,
POP,	= POP JMP,		

HANDY REFERENCE CARD

valFORTH™

SOFTWARE SYSTEM

EDITOR 1.1 COMMAND SUMMARY

Below is a quick reference list of all [illegible] recognizes.

Entering the Edit Mode [illegible]

V	(scr# ---)	* [illegible] the specified screen.
L	(---)	* [illegible]
WHERE	(---)	* [illegible] the cursor over [illegible] error.
LOCATE cccc	(---)	[illegible] the cursor over [illegible]
LOCATOR	(ON/OFF ---)	[illegible] the next [illegible] command above.
#BUFS	(#lines --)	[illegible] buffer. [illegible]

Cursor Movement (issued within [illegible])

ctrl	↑	* [illegible] the bottom line [illegible]
ctrl	↓	* [illegible]
ctrl	←	* [illegible] wrapping to the [illegible]
ctrl	→	* [illegible]
RETURN		[illegible]
TAB		[illegible]

Editing Commands [illegible]

ctrl	INS	[illegible]
ctrl	DEL	[illegible]
shift	INS	* [illegible]
shift	DEL	* [illegible]
ctrl	[illegible]	[illegible]
BACKS		* [illegible]
ctrl	[illegible]	[illegible]

Buffer Management [illegible]

ctrl	[illegible]	[illegible]
ctrl	F	[illegible]
ctrl	[illegible]	[illegible]
ctrl	[illegible]	[illegible]
ctrl	R	[illegible]
ctrl	B	[illegible]
ctrl	C	[illegible]

*Note: The current buffer line [illegible]

Changing Screens [illegible]

ctrl	[illegible]	[illegible]
ctrl	[illegible]	[illegible]
ctrl	[illegible]	* [illegible]
ctrl	[illegible]	* [illegible]

Special Keys [illegible]

ESC		* [illegible]
ctrl	A	[illegible]
ctrl	[illegible]	[illegible]
ctrl	[illegible]	[illegible]

Screen Management [illegible]

FLUSH	(--)	* [illegible]
EMPTY-BUFFERS	(--)	* [illegible]
COPY	(from to --)	* [illegible]
CLEAR	(scr# --)	* [illegible]
CLEARS	(scr# #screens --)	[illegible]
SMOVE	(from to #screens --)	[illegible]

* EDITOR 1.0 COMMAND